apple II

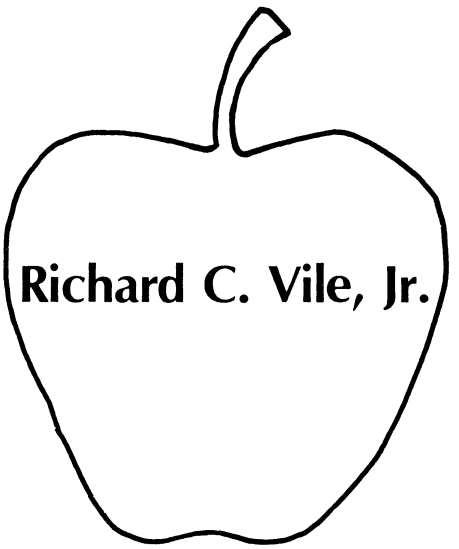
PROGRAMMER'S HANDBOOK

PROGRAMS AND PROGRAMMING TECHNIQUES IN
APPLE INTEGER BASIC, APPLESOFT BASIC, PASCAL,
AND 6502 ASSEMBLY LANGUAGE ▫ RICHARD VILE, JR.

RICHARD C. VILE, JR. is a senior software specialist for GemNet Software Corporation in Ann Arbor, Michigan. Active in the computer field since 1974, Mr. Vile has taught mathematics and computer science at Eastern Michigan University, worked as a programmer, and written articles for several leading computer magazines.

apple II

PROGRAMMER'S HANDBOOK



PROGRAMS AND PROGRAMMING TECHNIQUES
IN APPLE INTEGER BASIC, APPLESOFT BASIC,
PASCAL, AND 6502 ASSEMBLY LANGUAGE

GRANADA
London Toronto Sydney New York

Granada Technical Books
Granada Publishing Ltd
8 Grafton Street, London W1X 3LA

ISBN 0-246-12027-4

First published in the United States of America 1982 by
Prentice-Hall, Inc.
First published in Great Britain 1983 by Granada Publishing Ltd.
Reprinted 1983
Reprinted 1984

Printed in Great Britain by
Richard Clay (The Chaucer Press) Ltd,
Bungay, Suffolk.

Contents

Preface: How to Use This Book

This is a book *of* and *about* software. It emphasizes the use of four programming languages available on the APPLE II personal computer:

- Integer BASIC
- APPLESOFT BASIC
- APPLE (UCSD) Pascal
- APPLE (6502) Assembler

It is a book *of* software since there are over 40 programs whose source text have been printed in their entirety in the book. These are *complete* programs, tested and ready to use—such as you might buy in your local computer store. They include such applications as:

- Graphics

  Video—an animated TV test pattern.
  Alphabetics—a collection of entertaining and educational programs for kids.
  Dazzle—dynamic abstract low resolution graphics.

- Education

  Duelling DO Loops—interactive fun with BASIC FOR statements.
  APPLE Trivia—several versions of a configurable quiz program.

- Utilities

  BASIC subroutine loader.
  Pascal Units—calendar, low-res graphics, etc.
  An APPLESOFT Text Editor.

- Languages

  Hex calculator interpreter.
  Scanning programs.

- Entertainment

  The Fifteen Puzzle—interactive game and intelligent program versions.
  The Analogies Generator—nonsense phrases from the computer.

Some programs appear in more than one version. This allows comparison of the strengths and weaknesses of the languages in use. All of the programs may be modified and this is in fact encouraged. That's one of the reasons

why we say the book is *about* software. In addition to the programs themselves, there is ample commentary concerning the features and design of many of the programs. There is discussion about the language constructs used in the programs. Plus there are over 50 examples of what we have chosen to call APPLE Programming Tips.

A programmer is one who programs

Each programming tip is highlighted in the text by a banner like the one above and a brief descriptive title, such as "A programmer is one who programs." This saying exemplifies the philosophy of the book. We believe that in order to learn about software or programming, you must engage in programming yourself.

Read other peoples' programs

One excellent way to learn about programming is to read other peoples' programs. That is why we have included complete and substantial programs. To get the most out of the book, you should *read and study* the programs, as well as run them.

That is not to say that we don't believe in the *mechanics* of programming. This is not our aim in this particular book. We will assume that you have learned or are learning about the languages from one of the many excellent introductory texts on language rules and regulations that are available. We hope to take you a step beyond those tutorials, however, and show you how programs are constructed.

Organize your programs for understanding

Since this book is for relative beginners to the art of programming, yet wishes to present substantial examples of real programs, the programs included here have been written with the conscious assumption that people will *read* them. Consequently, the keyword in their construction has been *clarity*. Techniques to make the programs *more efficient* have been avoided in most cases, since that would have made them harder to understand. We hope that most programs in this book can be read and understood using a top-down approach. That is, by first absorbing the higher level structure and then gradually working down into the details. Most of the programs presented use many subroutines or procedures which make this process easier.

The book has been divided into four sections corresponding to the four languages mentioned above. There is some cross referencing of programs. For example, the Amper-Letters program of Chapter 11 uses some assembly language routines from Chapter 19. The APPLE Trivia Quiz program is reincarnated in APPLESOFT and Pascal after being introduced in Integer BASIC.

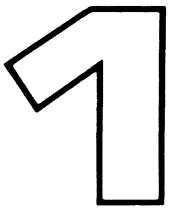
Explore!

At the end of most Chapters there is a section entitled "Explorations." In these sections, we present suggestions for actual programming. These range from the trivial to the extremely difficult. They give you your chance to practice what we have preached throughout the book. They are called Explorations because of their somewhat open-ended nature. We hope that they will lead you to experiment with ideas and modifications of your own. Experimenting with new ideas, modifying programs step by step, trying out new techniques: all these activities will lead you to a better understanding of programming and to more enjoyment of your APPLE.

Happy programming and enjoy!!

Dick Vile

Chapter



APPLE Integer BASIC

In this chapter, we give a very brief summary of Integer BASIC, indicating some of its desirable features and their usefulness. This is not intended to supplant either APPLE's tutorial manual on the subject or any other tutorial materials. We assume that the reader is familiar with some dialect of BASIC, probably Integer BASIC, and do not pretend that this is a beginner's guide. Many of the points we make in this section will be amplified upon in later chapters.

INTRODUCING INTEGER

Integer BASIC was the first APPLE language after 6502 machine language. Folklore has it that Steve Wozniak implemented the entire interpreter using only the mini-assembler of the original Monitor ROM as a tool—truly an impressive feat.

Integer BASIC is a somewhat stripped down version of the BASIC language: it does not support floating-point variables or functions, has limited string capabilities, provides no user-defined functions, and lacks multidimensional arrays. Nonetheless it is a useable language and provides some features not commonly found in many implementations of BASIC: arbitrary length identifiers, support for (low-resolution) color graphics, access to game paddle inputs, memory-mapped display support, the ability to CALL machine language routines, etc.

Playing Games

Integer BASIC provides enough features to allow interesting games to be written. In particular it allows color graphics displays to be produced in sixteen different colors on the low-resolution graphics display. The programmer is given the ability to plot points, draw horizontal and vertical lines, and to change colors. Of course, there are statements to allow changing from text to graphics modes and vice-versa.

Integer BASIC Graphics Commands

```
PLOT COL,ROW
HLIN COL1,COL2 AT ROW1
VLIN ROW1,ROW2 AT COL1
COLOR = N
GR
TEXT
```

These commands are pretty simple, yet comprehensive enough to allow fairly sophisticated graphics to be produced. Some of these capabilities are demonstrated in Chapters 3, 4, and 6.

In addition to the color graphics capabilities, Integer BASIC programs have access to two or more game paddle inputs via the built-in function PDL.

PDL(n)

returns the current reading of the *n*th game paddle, where *n* is between 0 and 3. The standard APPLE comes equipped with two paddles corresponding to $n = 0$ and $n = 1$. Most Integer BASIC programs make do with these two, although it is possible to add two more paddles to the APPLE.

The APPLE II has a built-in speaker. This is not supported directly by a statement like PDL, but it is possible to access the speaker using the PEEK or POKE functions. The memory location numbered −16336 is "tied" to the speaker in such a way that every time it is referenced as the argument of a PEEK or a POKE, the speaker makes a tiny click. Playing around with various combinations of such references in loops, with delays in between will produce various noises and tones. This is illustrated by the following brief program. Try running it and twisting the PDL(1) control back and forth very slowly. With some experimentation, you will see why it has been dubbed the "squeaky-door" demo. See Listing 1.1 at the end of this chapter.

Display Features

The APPLE II computer provides a memory-mapped display. This means that an area of memory is dedicated to storing the information that appears on the screen. Whenever one of these memory locations is changed, the difference is instantaneously visible on the screen. Integer BASIC takes some advantage of this design by allowing the cursor on the screen to be positioned to any desired row and column before a PRINT statement is issued. This allows a program to selectively update portions of the screen without destroying other information on the remainder of the screen. The cursor positioning statements are:

VTAB row
TAB col

These are taken full advantage of in the APPLE Trivia quiz of Chapter 2, as well as in many other programs.

In addition to the VTAB and TAB commands, Integer BASIC programs can take advantage of other features of the display. In particular, the Monitor ROM routines are used by the interpreter to do screen output. Certain parameters which these routines make use of may be directly controlled by the use of POKE statements. In particular, the concept of a scrolling window is frequently made use of in Integer BASIC programs. We deal with this concept and its associated parameters in Chapter 2 as well.

Coding Advantages of Integer BASIC

Integer BASIC offers some significant advantages over many other versions of BASIC in writing clear and easily understood programs. Probably the biggest of these advantages is its rules regarding identifiers.

1. Integer BASIC variable names may be as long or as short as the programmer wishes.
2. Variable names may be used in place of line numbers in GOTO and GOSUB statements.

The use of long variable names which may be chosen to directly suggest the intended use of the variable in the program has significant psychological advantages when reading programs. Instead of having to remember, for example, that variable S1 represents a score, the programmer may simply use SCORE as the name of the variable.

The ability to give names to subroutines which the program calls is another significant advantage both in reading and understanding programs. Compare, for example:

```
100 GOSUB 1000          100 GOSUB
                            INITIALIZE
105 GOSUB 9000          105 GOSUB
                            INTRODUCTION
110 GOSUB 5000          110 GOSUB
                    with    PLAYGAME
120 GOSUB 8000          120 GOSUB
                            ASKREPEAT
125 IF A$="YES"         125 IF ANSWER$=
    THEN 110                "YES" THEN 110
130 END                 130 END
```

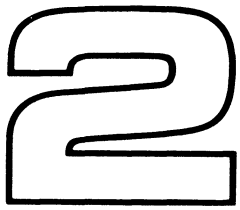
To use Integer BASIC most effectively, a programmer should take full advantage of these capabilities, choosing names which are suggestive and which will lead to a better understanding for those who will read the programmer's code—that includes the programmer her/himself as well. We try very hard to practice what we preach in all the Integer BASIC programs in this book.

LISTINGS

LISTING 1.1 SQUEAKY DOOR DEMO

```
>LIST
    5 SPKR=-16336
   10 X= PEEK (SPKR)+ PEEK (SPKR)
   15 REM ============================
   16 REM = VARY THE FOLLOWING DELAY =
   17 REM = LOOP TO PRODUCE THE      =
   18 REM = SQUEAKY DOOR EFFECT.     =
   19 REM ============================
   20 FOR I=1 TO PDL (1): NEXT I
   30 GOTO 10

>
```

Chapter



Interactive Programs: Using Menus

A large number of APPLE II programs take the form of interactive dialogues with the user. A standard approach to the user interface is the use of *menus*. In this chapter we discuss techniques for effective, user-friendly interactive programs. In the process, we present an example of a menu-driven, interactive program, which illustrates many of the techniques discussed.

1. EFFECTIVE USE OF THE APPLE II SCREEN

The APPLE II features a *memory-mapped* display. This means that a portion of the APPLE's system memory is set aside to hold the information which appears on the screen. When a new value is stored into one of these locations, the difference is *instantaneously* visible on the screen. As a consequence, portions of the display may be selectively updated without affecting the rest.

Cursor Controls

The APPLE's text display holds 24 lines of 40 characters each. A BASIC program displays text on the screen by the use of PRINT statements. The output goes to the screen at the "current" row and column position. This position is remembered by BASIC as the program runs, and may be *modified* by the use of the VTAB and TAB statements.

VTAB This statement positions the screen cursor at the row whose number is the value of the expression following VTAB:

```
VTAB 23
VTAB I + 5
VTAB 2*J + 1
```

TAB This positions the screen cursor at the column whose number is the value of the expression following TAB:

```
TAB 39
TAB 3*I + 2
```

The VTAB and TAB statements operate independently of each other. In particular, the following sequence of statements:

```
TAB 20
VTAB 15
TAB 10
```

will leave the cursor on row 15 and column 10. In some systems, the fact that the cursor started at column 20 when the TAB 10 statement was executed would cause the row position to advance to 16. This is *not* the case in APPLE Integer BASIC.

The limits on TAB and VTAB are:

TAB: 1–40

VTAB: 1–24

Any attempts to go beyond these limits will earn you a severe reprimand from the Integer BASIC Interpreter. The production of pleasing and effective displays will rely heavily on the use of TAB and VTAB.

Monitor ROM Support Routines

Routines in the APPLE Monitor ROM or Autostart ROM control the flow of information to the screen. We will make use of several of these on a regular basis—mainly to erase parts of the screen. ROM routines are accessed via the CALL statement:

CALL constant or **CALL IDENTIFIER**

For example, CALL –936 will cause the text screen to be erased and the cursor to return to the upper left-hand corner of the screen. The routines we shall rely on most are summarized in Table 2.1. These routines can be quite helpful in maintaining a neat screen appearance.

Highlighting—Use of Inverse Mode

The use of inverse video can also be quite effective in highlighting or emphasizing various portions of a screen display. The use of inverse video from Integer BASIC takes on a slight aspect of "magic" in that it is supported by means of a POKE statement. The memory location numbered 50 is used in a special way in producing the APPLE II text output. Depending on what value is stored there, the characters output to the screen may be normal, inverse, or blinking. Actually, it's more complicated than that, but we won't go into it just here. All we need for now is a way to turn inverse video on and off:

```
POKE 50,63    :REM SELECT INVERSE MODE
POKE 50,255   :REM SELECT NORMAL MODE
```

Thus, the short program:

```
10 CALL -936
20 VTAB 10: TAB 15
30 POKE  50,63: PRINT "INVERSE";
40 POKE 50,255: PRINT "NORMAL"
50 END
```

will clear the screen and print the word INVERSE in inverse mode on line 10 of the screen followed by the word NORMAL in normal mode.

The Scrolling Window

The APPLE II ROM routines support the idea of a *scrolling window* for the display screen. How all the various screen support goodies interact with this concept is a little tedious, so we will try to introduce it a little at a time, coming back to it again in future sections and chapters.

The idea of a window is simple: it is a rectangular portion of the display screen which may be thought of as a subscreen within the entire screen. This is illustrated in Figure 2.1.

A window may be specified by naming four pieces of information:

Window left	the leftmost column of the window
Window width	the width in columns of the window

TABLE 2.1

Machine Language Routine	*ROM Address*		*Function*
	Hex	*Decimal*	
HOME	FC58	–936	Clear the text screen and home the cursor
CLREOL	FC9C	–868	Clear the screen from the current cursor position to the end of the line
CLREOP	FC42	–958	Clear the screen from the current cursor position to the end of the screen

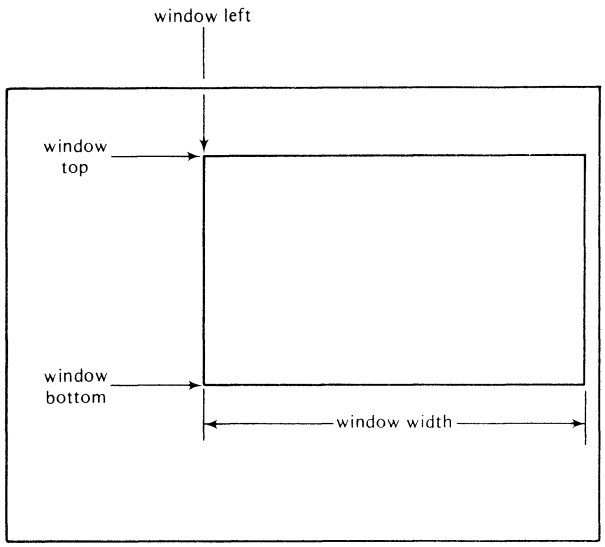


FIGURE 2.1 A Screen Window

Window top	the topmost row in the window
Window bottom	the bottommost row in the window

In APPLE's implementation of the scrolling window, these four items are remembered in specific APPLE RAM locations within the first 256 memory locations—the so-called "Page Zero" locations. Integer BASIC allows the programmer explicitly to store values into any RAM locations desired using the POKE statement. This means that the whereabouts of the window is under the BASIC programmer's control. The magic locations are as follows:

Window left	20 (hex) or 32 (decimal)
Window width	21 (hex) or 33 (decimal)
Window top	22 (hex) or 34 (decimal)
Window bottom	23 (hex) or 35 (decimal)

Once the window parameters have been set, scrolling will be limited to the text inside the window. Put another way, any text *outside* the window will not budge when scrolling takes place.

Scrolling

When a line of text is printed at the bottom of the window, all lines of text move up one. The top line disappears out of the window and when the bottom line moves up, it makes room for more text to appear where it used to be.

The window parameters do not control *all* placement of text, however. VTAB statements work regardless of the window top and window bottom. TAB statements are made relative to window left and they do respect the window width.

Try the following program:

```
 1 VTAB 10: TAB 10:PRINT "**********"
 2 FOR I= 1 TO 10:TAB 10:PRINT
   "*          *":NEXT I
 3 TAB 10: PRINT "**********"
10 POKE 32,10:POKE 33,10
12 POKE 34,10:POKE 35,20
25 VTAB 1:TAB 5:PRINT "LINE 1 COL 5"
26 VTAB 10:TAB 25:PRINT "HI"
27 VTAB 23:TAB 5:PRINT "23,5"
```

This demonstrates two facts. VTAB allows you to violate the vertical extent of the window. TAB does not allow you to violate the horizontal. Why all this works the way it does is a consequence of the way the Monitor ROM routines were written. We shall not go into all the details here.

2. THE SKELETON OF A MENU-DRIVEN PROGRAM

Before plunging into a full-length sample program, we shall discuss the idea of a *skeleton* program: an outline which will serve to guide the development of many different real programs. In order to develop the skeleton program, we need to discuss menu programs in general.

What is a Menu?

A *menu* is a list of choices, presented to the user of a program, usually on the video display. The user selects a menu item for the program to execute and the program transfers control to the appropriate section. A choice is made by pressing a key which corresponds to one of the menu selections—this could be a number or letter or even a special character—that depends on how the menu is presented.

Choosing a Personal Menu Style

Menu displays may range from the simple to the ornate. There is considerable scope here for creativity and original application of the basic screen manipulation tools.

The simplest presentation is a numbered or lettered list of short titles or phrases, each of which describes one of the options of the program. The program accepts the user's choice by reading a number or letter.

Information about what the program does or about individual choices in the menu may adorn the screen in addition to the bare text of the choices themselves.

3. CODING THE MENU-DRIVER

There are two basic facets to a menu-driver:

- Presenting the list of choices.
- Determining the user's choice and invoking the appropriate section of support in the program.

Presenting the list of choices is pretty much a matter of personal style. You will get to see one approach in the Trivia program later on. The only constraint we follow is that all the choices should fit on a single screen. If this constraint is violated, then your program is a candidate for "quick weight loss"!

Menu choices

Most menus are presented as a *numbered* or *lettered* list of choices. The program user may respond to the menu by one or two simple keystrokes. To convert this to a number which may be used to "vector" the program to the appropriate supporting code, the Integer BASIC function ASC may be used. We give the details following the presentation of the sample program.

Use skeleton programs

A *program skeleton* is a general outline from which may grow many different programs of the same nature. The skeleton may be nothing more than a standard outline which dictates what functional pieces will fit where. It may also contain a few or many *standard subroutines* which are used in many different programs. Examples of this in the APPLE Trivia program are WAIT and GET.

Figure 2.2 illustrates one possible bare-bones program skeleton for menu-driven programs. It indicates seven functional subdivisions of the program and a suggested range of line numbers to use for each.

The distribution of the line numbers is flexible and depends on the amount of program to be devoted to each subdivision. The program of Listing 2.1 is a good example of a medium-size, interactive program, written in Integer BASIC. It provides a menu with a choice of several quizzes:

(A) CHESS HISTORY
(B) OLYMPICS
(C) BOOKS AND AUTHORS
(D) COMPUTER LORE
(E) GENERAL INFORMATION

0 – 9	Banner: Title and Author
10 – 99	Declarations
100 – 999	Menu
1000 – 3999	Support Subroutines
4000 – 29999	Subject Matter Subroutines
31000 – 31999	Initializations
32000 – 32999	Introduction

FIGURE 2.2 The Skeleton of a Menu Driven Program

The section on general information has been intentionally left blank—you may fill it in with your own trivia questions.

The program's introduction indicates that there are quizzes of the following types:

1. Matching
2. Multiple Choice
3. Short Answer
4. True or False

The only type which is actually implemented by the program so far is matching. You will be given the chance to try your hand at implementing one or more of the others in the Explorations section at the end of this chapter.

Matching is relatively simple to do, since there can be no doubt about the correctness of the answers. In our case, the key to this may be found in the subroutine which scores the quiz (MARK = 1550):

```
IF ANSWERS (ORDER(I)) = I THEN RIGHT =
RIGHT + 1
```

What this means may be explained briefly as follows:

> The order in which the answers are displayed on the screen is captured in the array ORDER. Thus, ORDER(I) contains the answer to the Ith question and is printed in the ORDER(I)th position. Therefore, the ANSWER to the ORDER(I)th question is I. Confused? Think about it for a while—then if you are still confused, draw a picture or two of possible orders.

(For more info, study the following subroutines: MIXUP = 1050, SHOWANSWERS = 1100, and GETRESPONSE = 1150).

Multiple choice and true or false should also be fairly easy to implement, since the answers are exact. On the other hand, short answer quizzes must be imperfect at

best—the person using the program is free to enter many different short answers, all of which may in some sense be correct. The program must not only check for the "pure" answer, it must *judge* whether a given answer should be considered a *match* for the pure answer. In general, large computer-based instruction systems contain considerable amounts of intelligence aimed at the resolution of this problem.

Study the APPLE Trivia program, shown in listing 2.1, to see what you can discover for yourself. Then read the next section for a number of explanatory *programming tips* about it. Figure 2.3 shows a diagram with all the subroutines of APPLE Trivia and how they depend on or are called from one another. Such a diagram is a good reference to have when studying a program of more than one or two pages in length.

4. INTEGER BASIC CODING STYLE

There are a number of practices which can make your life considerably easier when using Integer BASIC. Consistently applying these techniques should lead to programs which are easier to read and understand. This becomes significant when someone besides the original author attempts to modify and extend the program, or when the original programmer comes back to the program after a long absence.

In this section we give a number of programming tips which explain some of the programming practices (and in some case programming tricks) which have been used in coding the APPLE Trivia program.

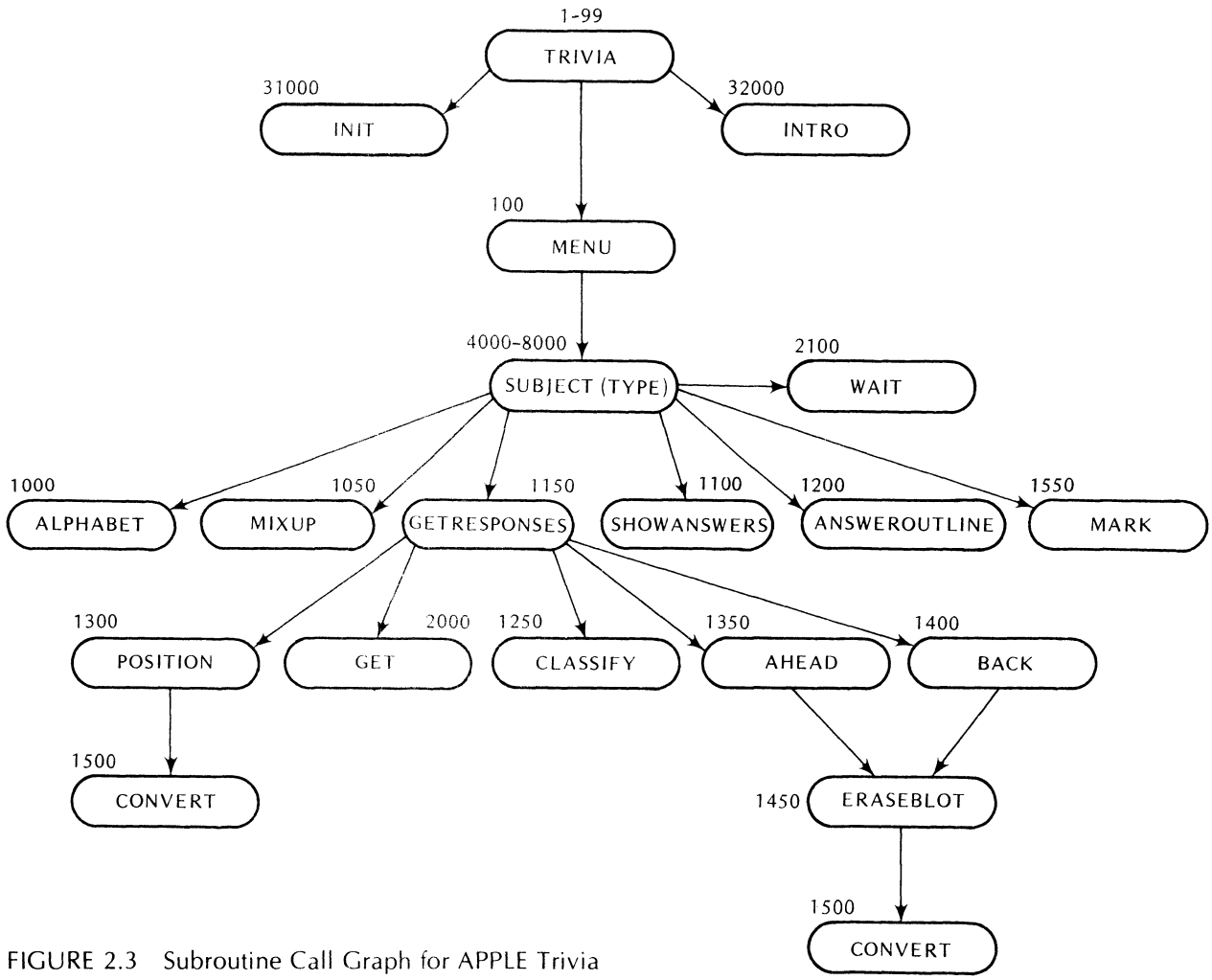


FIGURE 2.3 Subroutine Call Graph for APPLE Trivia

Use comment banners

The BASIC programming language has no way to indicate syntactically where a subroutine begins or ends. For this reason, it may be difficult to perceive the structure of a program visually. One way to remedy the situation is to use "comment banners" at the head of each subroutine. These banners could contain the name of the subroutine outlined by a punctuation mark that stands out:

```
1000 REM ================
1001 REM = ALPHABET SUBROUTINE =
1002 REM ================
```

This has a striking effect. Programs suddenly become much clearer and easier to absorb. The eye can easily isolate an entire subroutine at a glance, allowing the mind to concentrate solely on the statements of the program and not on its surroundings. Try this and see if you don't agree that it provides a significant psychological advantage.

Use more subroutines

When in doubt, use more subroutines rather than fewer. Subroutines seldom obfuscate and frequently clarify a program. Each subroutine should have a single well-defined task to carry out. A good example of this philosophy may be seen in the subroutine GETRESPONSE and its "subordinates." The typical beginning BASIC "hotshot" would probably code this as a single large subroutine. The major advantage that this yields is one of speed. BASIC does tend to be slow. On the other hand, putting this all into one chunk of code would make the program a great deal more difficult to read and what's worse, more difficult to modify.

Programs almost never remain unmodified. Programmers are by nature an inquisitive and inventive lot. They can't help themselves—they *must* modify programs. But after a program reaches a certain *critical mass,* this becomes difficult to do. A subroutine may be thought of as a program within a program. Making many of them of necessity keeps them small and intellectually manageable. Try it—you'll like it!

Name your subroutines

Integer BASIC allows the programmer to write:

GOSUB IDENTIFIER

It also allows identifiers of arbitrary length. Take advantage. Give your subroutines names which suggest their function and then call them with the GOSUB IDENTIFIER construct. For example:

```
GOSUB INTRO
GOSUB INIT
GOSUB MAIN
GOSUB CLEANUP
```

is much easier to understand than:

```
GOSUB 10000
GOSUB 9000
GOSUB 5000
GOSUB 20000
```

This is especially useful when you come back to a program after a long period of time of not having looked at it.

Use a declaration section

If you make use of Integer BASIC's long identifier capability, then it is a good idea to collect all subroutine name definitions and constant name definitions into one section of your program.

The natural place to put such a section is right at the beginning of the program where it may be easily referenced. Lines 10–99 usually prove adequate for this purpose.

In APPLE Trivia, lines 10–89 form the declaration section. In this section, names are given to:

Machine language routines **CLREOL = -868: HOME = -936,** etc.

Numeric constants **KBD = -16384 : CLR = -16368 : NUMSUB = 5**

String constants **D$ = "" : REM CONTROL-D**

BASIC subroutines **GET = 2000 : INTRO = 32000 : Wait = 2100,** etc.

NOTE: When using this approach, the use of RENUMBER utilities (such as that in APPLE's Programmer's Aid ROM) is not recommended. The reason is that most (if not all) such utilities will *not* change the values in statements such as GET=2000:INTRO=32000. The location of GET on the other hand will probably change, which means that GOSUB GET will probably wreak havoc.

How to sample the keyboard

What do you do when you want to find out the state of the APPLE keyboard?

- Has a key been struck?
- What is the numeric value of the input key?

The APPLE II dedicates the following memory locations:

-16384
-16368

for these purposes. The following short routine shows their use:

```
110 KEY=PEEK(-16384):IF KEY<128 THEN 110
120 POKE -16368,0
149 RETURN
```

From this routine, we may deduce:

- If the value in -16384 is less than 128, then no key has been struck.
- If the value in -16384 is greater than 128, then a key has been struck and its ASCII value is the contents of -16384 minus 128.
- In order to *clear* the keyboard to allow a new key to be sensed, the location -16368 must be POKEd—any value will do.

The magic locations are easier to remember if names are chosen for them. For example:

```
KBD = -16384
CLR = -16368
```

Then the standard GETKEY subroutine becomes:

```
100REM=======
101REM= GETKEY =
102REM=======
110 KEY =-PEEK(KBD):IF KEY<128 THEN 110
120 POKE CLR,0
149 RETURN
```

Use standard subroutines

Many functions of interactive programs are required in more than one program. It makes sense to create *standard* subroutines to perform these functions. For example, the GETKEY function described above. The temptation for most programmers is to combine this function with the part of the program which decides *what to do* with the values detected. That has the disadvantage that you must recode the GETKEY part each time you write a program, in order for it to fit in with the rest of the program. If you make it into a standard subroutine, then you may keep a notebook of such routines and merely copy it into your program. This saves time and allows you to concentrate on the unique parts of each new program you write.

Be discreet with tricks

Every programmer has at one time or another used tricks in writing programs. These are special coding practices that may range from the slightly nonobvious to the downright obscure. Because they make programs harder to decipher, they should be used with care and should always be explained somewhere. There may be disagreement about exactly what constitutes a trick, but a good rule might be "If it's doubtful, it's tricky."

Here are some of the tricks used in APPLE Trivia.

Use of the GOSUB statement

Each subject matter quiz is implemented by a different subroutine. These subroutines have *not* been named in the program (ignoring the tip given earlier). Rather, the line numbers where each of them starts have been stored in an array called SUBJECTS. See lines 31020–31024 of the program. Then the statement

```
GOSUB SUBJECTS(TYPE)
```

allows us to call any of the subject matter subroutines. If more subjects are added later, this part of the program will not have to be modified. The value of the variable TYPE determines which subject is to be taken.

Use of the ASC function

The variable TYPE is set as follows:

If the user keys	*Then TYPE is set to*
A	1
B	2
C	3
D	4
E	5

This bit of legerdemain is accomplished using the ASC function. The statement:

```
TYPE = KEY - ASC("@")
```

causes exactly the values indicated above to be set into TYPE. This technique is generally applicable whenever a single key value must be *converted* to another value.

Use of GOTO expression

In GETRESPONSE, the action depends on the CLASS of the user's input, which is determined by the subroutine CLASSIFY. The statement:

```
GOTO 1160 + CLASS*10
```

causes line 1160, 1170, 1180, or 1190 to be executed, depending on the value of CLASS (= 0, 1, 2, or 3). This serves to replace the use of ON GOTO as in APPLESOFT BASIC.

5. EXPLORATIONS

- Implement one or more of the other types of quizzes. Some possibilities are:

  Multiple Choice
  True or False
  Short Answer

  If you try the short answer quiz you will probably have to make some restrictions in order to judge the answers effectively.

- Experiment using windows. Here are some possibilities:

  Keep the menu in a window in one portion of the screen. Switch between the menu window and the remainder of the screen during quizzes. Highlight the choice in the menu using inverse video.

  Outline the windows with asterisks or equals signs for emphasis.

  Figure out how to maintain more than one window at a time—flip-flop between them.

- Other menu programs.

  Design other menu-driven programs of your own.

  Use some of the programming tips of this chapter.

- Improving existing programs.

Do you have programs already that might be partially or wholly rewritten using some of the techniques presented in this chapter?

LISTINGS

LISTING 2.1 APPLE TRIVIA QUIZ

```
]LIST
  1 REM ============================
  2 REM =                          =
  3 REM =    APPLE TRIVIA QUIZ     =
  4 REM =            BY            =
  5 REM =DR. RICHARD C. VILE, JR.=
  6 REM = ALL COMMERCIAL RIGHTS  =
  7 REM =       RESERVED           =
  8 REM =                          =
  9 REM ============================
 10 D$="": REM  CONTROL-D
 15 KBD=-16384:CLR=-16368:CLREOL=
    -868:CLREOP=-958:HOME=-936:
    BELL=-198
 16 GET=2000:INTRO=32000:WAIT=2100

 17 QUIT=5:ALPHABET=1000:INIT=31000

 18 MIXUP=1050:SHOWANSWERS=1100
    :NUMSUB=5
 19 RESPONSELINE=18:GETRESPONSES=
    1150
 20 ANSWEROUTLINE=1200:CLASSIFY=
    1250
 21 POSITION=1300:AHEAD=1350:BACK=
    1400
 22 ERASEBLOT=1450:CONVERT=1500
    :MARK=1550
 23 TOOT=1600:DELAY=25:QUIZ=1700

 24 BANNER=1750
 50 DIM ORDER(10),ANSWERS(10)
 51 DIM ALPH$(26),SUBJECTS(5)
 52 DIM TAKEN(5)
 90 GOSUB INIT: GOSUB INTRO
100 CALL HOME: POKE 50,255
102 GOSUB BANNER: VTAB 12: TAB
    5
105 PRINT "CHOOSE ONE OF THE FOLLOWI
    NG:"
110 PRINT
120 TAB 5: PRINT "(A) CHESS HISTORY"

125 TAB 5: PRINT "(B) OLYMPICS"

130 TAB 5: PRINT "(C) BOOKS AND AUTH
    ORS"
135 TAB 5: PRINT "(D) COMPUTER LORE"

140 TAB 5: PRINT "(E) GENERAL INFORM
    ATION"
145 TAB 5: PRINT "(F) EXIT"

195 GOTO 200
199 CALL BELL
200 GOSUB GET
205 TYPE=KEY- ASC("A")
210 IF (TYPE<0) OR (TYPE>5) THEN
    199
215 IF TYPE#QUIT THEN 250
220 CALL HOME: END
250 IF NOT TAKEN(TYPE+1) THEN 290

255 VTAB 18: CALL CLREOP
260 TAB 1: PRINT "SORRY BUT YOU HAVE
     ALREADY TAKEN THAT"
265 PRINT "QUIZ.  YOU WILL HAVE TO T
    RY ANOTHER"
270 PRINT "SUBJECT."
275 GOSUB WAIT
280 GOTO 100
290 GOSUB SUBJECTS(TYPE+1)
299 GOTO 100
1000 REM ========================
1001 REM = ALPHABET SUBROUTINE =
1002 REM ========================
1005 VTAB BASE: TAB 20
1010 PRINT "A."
1011 TAB 20: PRINT "B."
1012 TAB 20: PRINT "C."
1013 TAB 20: PRINT "D."
1014 TAB 20: PRINT "E."
1015 TAB 20: PRINT "F."
1016 TAB 20: PRINT "G."
1017 TAB 20: PRINT "H."
1018 TAB 20: PRINT "I."
1019 TAB 20: PRINT "J."
1049 RETURN
1050 REM ====================
1051 REM = MIXUP SUBROUTINE =
1052 REM ====================
1055 FOR I=1 TO 10:ORDER(I)=-1: NEXT
     I
1060 FOR I=1 TO 10
1065 J= RND (10)+1
1070 IF ORDER(J)#-1 THEN 1065
1075 ORDER(J)=I
1080 NEXT I
1099 RETURN
1100 REM ============================
1101 REM = SHOWANSWERS SUBROUTINE =
1102 REM ============================
1110 FOR I=1 TO 10
1115 WHO=ORDER(I)
1117 ANSBASE=BASE+I-1
1120 GOSUB FIRSTANSWER+(WHO-1)*10
```

```
1125 NEXT I
1149 RETURN
1150 REM ============================
1151 REM = GETRESPONSES SUBROUTINE =
1152 REM ============================
1154 FOR I=1 TO 10:ANSWERS(I)=0:
      NEXT I
1155 GOSUB POSITION: GOSUB GET: GOSUB
     CLASSIFY
1156 GOTO 1160+CLASS*10
1160 REM *****  CLASS = 0  *****
1162 INDEX=KEY- ASC("@"):A$=ALPH$
     (INDEX,INDEX)
1163 PRINT A$;:ANSWERS(SPOT)=INDEX
1164 GOSUB AHEAD: GOSUB POSITION
1165 GOTO 1155
1170 REM *****  CLASS = 1  *****
1172 IF KEY=149 THEN GOSUB AHEAD
1173 IF KEY=136 THEN GOSUB BACK
1175 GOTO 1155
1180 REM *****  CLASS = 2  *****
1182 GOSUB ERASEBLOT
1185 RETURN
1190 REM *****  CLASS = 3  *****
1192 GOSUB TOOT
1195 GOTO 1155
1199 RETURN
1200 REM ==================
1201 REM = ANSWER OUTLINE =
1202 REM ==================
1210 VTAB RESPONSELINE: TAB 1
1215 PRINT "1.     2.     3.     4.
       5.     "
1220 PRINT "6.     7.     8.     9.
      10."
1225 PRINT : PRINT "MOVE CURSOR WITH
     RIGHT AND LEFT ARROWS"
1230 PRINT "HIT ENTER WHEN FINISHED"

1249 RETURN
1250 REM ======================
1251 REM = CLASSIFY SUBROUTINE =
1252 REM ======================
1255 CLASS=3
1260 IF (KEY=136) OR (KEY=149) THEN
     CLASS=1
1265 IF (KEY>= ASC("A")) AND (KEY<=
      ASC("Z")) THEN CLASS=0
1270 IF KEY=141 THEN CLASS=2
1299 RETURN
1300 REM ======================
1301 REM = POSITION TO ANSWER =
1302 REM = BLANK SPOT.        =
1303 REM ======================
1305 GOSUB CONVERT
1310 VTAB RESPONSELINE+ROW
1315 TAB 7*COL+3
1320 POKE 50,63: PRINT " ";: POKE
     50,255
1349 RETURN
1350 REM ==============
1351 REM =    AHEAD    =
1352 REM ==============
1355 GOSUB ERASEBLOT
1356 SPOT=SPOT+1
1360 IF SPOT=11 THEN SPOT=1
1399 RETURN
1400 REM ===============
1401 REM =     BACK     =
1402 REM ===============
1410 GOSUB ERASEBLOT
1411 SPOT=SPOT-1
1415 IF SPOT=0 THEN SPOT=10
1449 RETURN
1450 REM ========================
1451 REM = ERASEBLOT SUBROUTINE =
1452 REM ========================
1455 GOSUB CONVERT
1460 VTAB RESPONSELINE+ROW
1465 TAB 7*COL+3: PRINT " ";
1499 RETURN
1500 REM ===============
1501 REM = CONVERT SPOT =
1502 REM ===============
1505 ROW=SPOT/6
1510 COL=(SPOT-1) MOD 5
1549 RETURN
1550 REM ================
1551 REM = MARK: SCORING =
1552 REM ================
1555 RIGHT=0
1560 FOR I=1 TO 10
1562 IF ANSWERS(ORDER(I))#I THEN
     1565
1563 RIGHT=RIGHT+1
1564 TAB 1: VTAB BASE+ORDER(I)-1
     : POKE 50,63: PRINT ORDER(I)
     ;".";: POKE 50,255
1565 NEXT I
1566 VTAB 21: TAB 1: CALL CLREOL
1567 PRINT : CALL CLREOL
1568 PRINT "YOU GOT ";
1570 FOR II=1 TO RIGHT: GOSUB TOOT:
      NEXT II
1572 PRINT RIGHT;" CORRECT."
1599 RETURN
1600 REM ==============
1601 REM =    TOOT      =
```

LISTING 2.1 (*cont.*)

```
1602 REM ===============
1605 PRINT "";: REM  CONTROL-G
1610 FOR I=1 TO DELAY: NEXT I
1649 RETURN
1700 REM ============================
1701 REM = QUIZ - DRIVER CALLED   =
1702 REM =         FROM INDIVIDUAL =
1703 REM =         TOPICS SECTIONS =
1704 REM ============================
1720 GOSUB QUESTIONS
1725 GOSUB ALPHABET
1730 GOSUB MIXUP
1735 GOSUB SHOWANSWERS: PRINT
1736 PRINT "**************************
     ***************"
1740 GOSUB ANSWEROUTLINE
1742 SPOT=1: GOSUB POSITION
1744 GOSUB GETRESPONSES
1745 GOSUB MARK
1748 GOSUB WAIT
1749 RETURN
1750 REM ==================
1751 REM =      BANNER      =
1752 REM ==================
1755 VTAB 1: TAB 1: PRINT "==========
     ================================"
     ;
1756 PRINT "=
                    =";
1757 PRINT "=           APPLE TRIVIA Q
     UIZ             =";
1758 PRINT "=
                    =";
1759 PRINT "=      BY DR. RICHARD C. V
     ILE, JR.        =";
1760 PRINT "=
                    =";
1761 PRINT "=            (C) MARCH 198
     1               =";
1762 PRINT "============================
     ==============";
1799 RETURN
2000 REM ======================
2001 REM = GET KEY SUBROUTINE =
2002 REM ======================
2005 KEY= PEEK (KBD): IF KEY<128
      THEN 2005
2010 POKE CLR,0
2049 RETURN
2100 REM ==================
2101 REM = WAIT SUBROUTINE =
2102 REM ==================
2105 POKE 50,63: VTAB 24: TAB 5:
      PRINT "PRESS ANY KEY TO CONTINU
     E";
2110 POKE CLR,0
2115 GOSUB GET
2120 POKE 50,255
2149 RETURN
4000 REM ====================
4001 REM =                  =
4002 REM =  CHESS HISTORY   =
4003 REM =                  =
4004 REM ====================
4010 CALL HOME: PRINT
4015 PRINT "MATCH THE NAMES OF THE FO
     LLOWING WORLD"
4016 PRINT "CHESS CHAMPIONS - LAST NA
     ME TO FIRST"
4017 PRINT
4020 QUESTIONS=4100:FIRSTANSWER=
     4200:BASE=5
4025 GOSUB QUIZ
4048 TAKEN(1)=1
4049 RETURN
4100 PRINT "1. CAPABLANCA"
4105 PRINT "2. MORPHY"
4110 PRINT "3. STEINITZ"
4115 PRINT "4. SPASSKY"
4120 PRINT "5. BOTVINNIK"
4125 PRINT "6. ALEKHINE"
4130 PRINT "7. ANDERSSEN"
4135 PRINT "8. FISCHER"
4140 PRINT "9. PETROSYAN"
4145 PRINT "10.SMYSLOV"
4149 RETURN
4200 VTAB ANSBASE: TAB 23
4202 PRINT "JOSE RAOUL"
4205 RETURN
4210 VTAB ANSBASE: TAB 23
4212 PRINT "PAUL C."
4215 RETURN
4220 VTAB ANSBASE: TAB 23
4222 PRINT "WILHELM"
4225 RETURN
4230 VTAB ANSBASE: TAB 23
4232 PRINT "BORIS"
4235 RETURN
4240 VTAB ANSBASE: TAB 23
4242 PRINT "MIKHAIL"
4245 RETURN
4250 VTAB ANSBASE: TAB 23
4252 PRINT "ALEXANDER"
4255 RETURN
4260 VTAB ANSBASE: TAB 23
4262 PRINT "ADOLF"
4265 RETURN
4270 VTAB ANSBASE: TAB 23
```

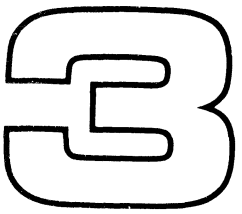
```
4272 PRINT "ROBERT"
4275 RETURN
4280 VTAB ANSBASE: TAB 23
4282 PRINT "TIGRAN"
4285 RETURN
4290 VTAB ANSBASE: TAB 23
4292 PRINT "VASSILY"
4295 RETURN
4999 RETURN
5000 REM ====================
5001 REM =                  =
5002 REM =   OLYMPICS       =
5003 REM =                  =
5004 REM ====================
5010 CALL HOME
5015 PRINT : PRINT "MATCH THE NAMES O
     F THE FOLLOWING "
5016 PRINT "OLYPIANS WITH THEIR EVENT
     S:"
5017 PRINT
5020 QUESTIONS=5100:FIRSTANSWER=
     5200:BASE=5
5025 GOSUB QUIZ
5048 TAKEN(2)=1
5049 RETURN
5100 PRINT "1. BOB SEAGREN"
5105 PRINT "2. PETER SNELL"
5110 PRINT "3. MARK SPITZ"
5115 PRINT "4. RAFER JOHNSON"
5120 PRINT "5. CAROL HEISS"
5125 PRINT "6. EMIL ZATOPEK"
5130 PRINT "7. VALERY BRUMEL"
5135 PRINT "8. AL OERTER"
5140 PRINT "9. SPYROS LOUES"
5145 PRINT "10.REX CAWLEY"
5149 RETURN
5200 VTAB ANSBASE: TAB 23
5202 PRINT "POLE VAULT"
5205 RETURN
5210 VTAB ANSBASE: TAB 23
5212 PRINT "MILE RUN"
5215 RETURN
5220 VTAB ANSBASE: TAB 23
5222 PRINT "SWIMMING"
5225 RETURN
5230 VTAB ANSBASE: TAB 23
5232 PRINT "DECATHLON"
5235 RETURN
5240 VTAB ANSBASE: TAB 23
5242 PRINT "FIGURE SKATING"
5245 RETURN
5250 VTAB ANSBASE: TAB 23
5252 PRINT "5000 METER RUN"
5255 RETURN
5260 VTAB ANSBASE: TAB 23
5262 PRINT "HIGH JUMP"
5265 RETURN
5270 VTAB ANSBASE: TAB 23
5272 PRINT "DISCUS THROW"
5275 RETURN
5280 VTAB ANSBASE: TAB 23
5282 PRINT "MARATHON RUN"
5285 RETURN
5290 VTAB ANSBASE: TAB 23
5292 PRINT "400 METER HURDLES"
5295 RETURN
5999 RETURN
6000 REM ====================
6001 REM =                  =
6002 REM = BOOKS AND AUTHORS =
6003 REM =                  =
6004 REM ====================
6010 CALL HOME: PRINT
6015 PRINT "MATCH THE NAMES OF THE FO
     LLOWING AUTHORS";
6016 PRINT "TO THE WORKS WHICH THEY W
     ROTE."
6017 PRINT
6020 QUESTIONS=6100:FIRSTANSWER=
     6200:BASE=5
6025 GOSUB QUIZ
6048 TAKEN(3)=1
6049 RETURN
6100 PRINT "1. J. CLAVELL"
6105 PRINT "2. P.G. WODEHOUSE"
6110 PRINT "3. J. B. CABELL"
6115 PRINT "4. G. ORWELL"
6120 PRINT "5. C. P. SNOW"
6125 PRINT "6. C. S. LEWIS"
6130 PRINT "7. J. STEINBECK"
6135 PRINT "8. J. DOS PASSOS"
6140 PRINT "9. J. LONDON"
6145 PRINT "10.R. PENN WARREN"
6199 RETURN
6200 VTAB ANSBASE: TAB 23
6202 PRINT "KING RAT"
6205 RETURN
6210 VTAB ANSBASE: TAB 23
6212 PRINT "PIGS HAVE WINGS"
6215 RETURN
6220 VTAB ANSBASE: TAB 23
6222 PRINT "JURGEN"
6225 RETURN
6230 VTAB ANSBASE: TAB 23
6232 PRINT "1984"
6235 RETURN
6240 VTAB ANSBASE: TAB 23
6242 PRINT "THE MASTERS"
```

LISTING 2.1 *(cont.)*

```
6245 RETURN
6250 VTAB ANSBASE: TAB 23
6252 PRINT "SURPRISED BY JOY"
6255 RETURN
6260 VTAB ANSBASE: TAB 23
6262 PRINT "SWEET THURSDAY"
6265 RETURN
6270 VTAB ANSBASE: TAB 23
6272 PRINT "MANHATTAN TRANSFER"
6275 RETURN
6280 VTAB ANSBASE: TAB 23
6282 PRINT "THE SEA WOLF"
6285 RETURN
6290 VTAB ANSBASE: TAB 23
6292 PRINT "NIGHT RIDER"
6295 RETURN
7000 REM ====================
7001 REM =                  =
7002 REM =  COMPUTER LORE   =
7003 REM =                  =
7004 REM ====================
7010 CALL HOME: PRINT
7015 PRINT "MATCH THE NAMES OF THE FO
     LLOWING"
7016 PRINT "COMPUTERS TO THEIR MANUFA
     CTURERS."
7017 PRINT
7020 QUESTIONS=7100:FIRSTANSWER=
     7200:BASE=5
7025 GOSUB QUIZ
7048 TAKEN(4)=1
7049 RETURN
7100 PRINT "1. LEVEL 6"
7105 PRINT "2. NOVA"
7110 PRINT "3. B5500"
7115 PRINT "4. PDP-8"
7120 PRINT "5. SIGMA-7"
7125 PRINT "6. 6600"
7130 PRINT "7. ALTO"
7135 PRINT "8. SPECTRA 70"
7140 PRINT "9. 1130"
7145 PRINT "10.1108"
7149 RETURN
7200 VTAB ANSBASE: TAB 23
7202 PRINT "HONEYWELL"
7205 RETURN
7210 VTAB ANSBASE: TAB 23
7212 PRINT "DATA GENERAL"
7215 RETURN
7220 VTAB ANSBASE: TAB 23
7222 PRINT "BURROUGHS"
7225 RETURN
7230 VTAB ANSBASE: TAB 23
7232 PRINT "DIGITAL"
7235 RETURN
7240 VTAB ANSBASE: TAB 23
7242 PRINT "SDS"
7245 RETURN
7250 VTAB ANSBASE: TAB 23
7252 PRINT "CONTROL DATA"
7255 RETURN
7260 VTAB ANSBASE: TAB 23
7262 PRINT "XEROX"
7265 RETURN
7270 VTAB ANSBASE: TAB 23
7272 PRINT "RCA"
7275 RETURN
7280 VTAB ANSBASE: TAB 23
7282 PRINT "IBM"
7285 RETURN
7290 VTAB ANSBASE: TAB 23
7292 PRINT "SPERRY-UNIVAC"
7295 RETURN
7999 RETURN
8000 REM ========================
8001 REM =                      =
8002 REM = GENERAL INFORMATION =
8003 REM =                      =
8004 REM ========================
8010 CALL HOME
8899 RETURN
31000 REM ====================
31001 REM = INITIALIZATIONS =
31002 REM ====================
31010 FOR I=1 TO 10:ORDER(I)=I: NEXT
      I
31015 ALPH$="ABCDEFGHIJKLMNOPQRSTUVWXY
      Z"
31020 SUBJECTS(1)=4000
31021 SUBJECTS(2)=5000
31022 SUBJECTS(3)=6000
31023 SUBJECTS(4)=7000
31024 SUBJECTS(5)=8000
31030 FOR I=1 TO NUMSUB:TAKEN(I)=
      0: NEXT I
31999 RETURN
32000 REM ====================
32001 REM =                  =
32002 REM =   INTRODUCTION   =
32003 REM =                  =
32004 REM ====================
32005 CALL HOME
32010 VTAB 5
32015 PRINT "  WELCOME TO THE APPLE TR
      IVIA QUIZ!"
32020 PRINT "YOU WILL BE GIVEN A CHOIC
      E OF SEVERAL"
32025 PRINT "TOPICS TO BE QUIZZED ON.
```

```
      EACH TOPIC"
32030 PRINT "WILL HAVE A TEN QUESTION
      QUIZ OF ONE"
32035 PRINT "OF THE FOLLOWING TYPES:"

32037 PRINT
32040 PRINT "     1. MATCHING"
32045 PRINT "     2. MULTIPLE CHOICE"

32050 PRINT "     3. SHORT ANSWER"

32055 PRINT "     4. TRUE OR FALSE"
32060 PRINT : PRINT "  YOU MAY TAKE AL
      L THE TIME YOU WISH"
32065 PRINT "TO ANSWER EACH QUIZ, BUT
      ONCE A GIVEN"
32070 PRINT "TOPIC HAS BEEN TAKEN, YOU
       MAY NOT "
32075 PRINT "RETURN TO IT."
32080 PRINT "GOOD LUCK!!!"
32095 GOSUB WAIT
32099 RETURN

>
```

Chapter



Low-Resolution Graphics

In this chapter, we introduce some of the entertaining things that may be done with APPLE's low-resolution (low-res) graphics. We shall just scratch the surface here, returning to the topic in later chapters on APPLESOFT, Pascal, and 6502 Assembler.

1. THE VIDEO PROGRAM

The program of Listing 3.1 uses the HLIN and VLIN statements to create a psychedelic, moving test pattern. The pattern is produced by the careful use of symmetry and the orderly progression from the edge of the screen to the center and back out again of a crosshatch pattern produced by the use of graphics statements.

The subroutines responsible for the moving patterns are:

TICTAC = 1000
QUADTAC = 1200
OCTTAC = 1400

Each plots crosshatch patterns in progression. Figure 3.1 illustrates the locations of the lines which are plotted. The symmetry is achieved by observing the obvious dividing points on the screen into halves and quarters both horizontally and vertically. The same approach would not work for a screen 37 (or any other odd number of) units on a side.

Use RND to add variety

The program chooses randomly between the three subroutines and allows each to be called a random number of times. The Integer BASIC function RND is used to accomplish this.

RND Function

RND(N) produces a number in the range 0 to N − 1. The distribution of the actual values produced varies randomly and will not be repeated on two successive program runs.

RND(0) will produce an error, usually >32767 ERR. Writing RND(0) in bald-face fashion is pretty easy to avoid. However, a sequence such as:

```
R1 = PDL(1)
COLOR = RND(R1)
```

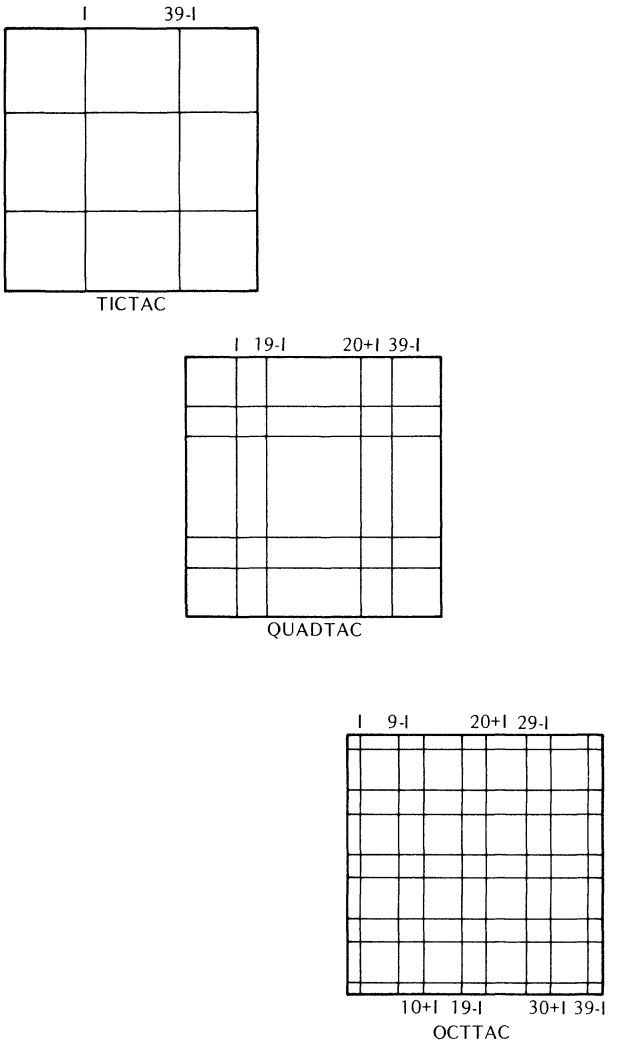


FIGURE 3.1 Video Program Plot Patterns

is apt to produce RND(0) in camouflage; i.e., if PDL(1) = 0. Protection against this may be obtained by always adding 1 to the argument of RND if it is a variable which could take on the value 0. For example:

```
COLOR = RND(R1+1)
```

In the VIDEO program, the use of RND combined with the use of the game paddles allows the user of the program to control:

- How fast the color changes.
- How often the program flips between TICTAC, QUADTAC, and OCTTAC.

To see how this works, study lines 100–185 and 1200–1249 of the program.

2. DRAWING FIGURES IN INTEGER BASIC

The HLIN, VLIN, PLOT, and COLOR statements give enough capability to BASIC to enable the programmer to design a variety of graphic effects. One possible application is "drawing" figures of various types at different places on the screen.

Absolute Addressing

When you wish to draw a static picture, that is, one in which each element will always appear at the same position on the screen, then it is appropriate to use *absolute addressing*. That means that the location of each HLIN, VLIN, and PLOT is given with constant values:

```
HLIN 10,20 at 15
VLIN 5,15 AT 3
PLOT 5,7
PLOT 13,3
etc.
```

This technique provides the fastest graphics production. It also provides the least flexible programming: if you change your mind about the location of the picture on the screen, you will have to rework every statement.

Variable Addressing

The various "-TAC" routines of the VIDEO program illustrate the more common approach: the use of *single variable names* to represent row and column values for graphics statements. For example:

```
HLIN 0,39 AT I
HLIN 0,39 AT 39-I
```

causes two horizontal lines to be plotted the full width of the screen, but located anywhere an equal distance from the top and bottom of the 40 × 40 graphics display. This approach to things allows a program to *calculate* values which determine the size and location of graphics displays. It is only slightly slower than the absolute addressing technique, but far more flexible.

Use base-offset graphics

In some graphics displays, it is desirable to be able to plot figures relative to a freely moving "origin." The same figure in size, shape, and color may be required at many different places on the screen. Even the variable address-

ing approach may have difficulty with such requirements. For example, suppose we wish to be able to draw the block letter B as shown in Figure 3.2.

Assuming that this letter is to be plotted in the upper left-hand corner of the screen, we might use the following statements:

```
HLIN 0,3 AT 0
HLIN 0,3 AT 3
HLIN 0,3 AT 6
VLIN 0,6 AT 0
VLIN 1,2 AT 4
VLIN 4,5 AT 4
```

On the other hand, if the letter is to be plotted with its upper left-hand corner located at row 5, column 10, we would need to use:

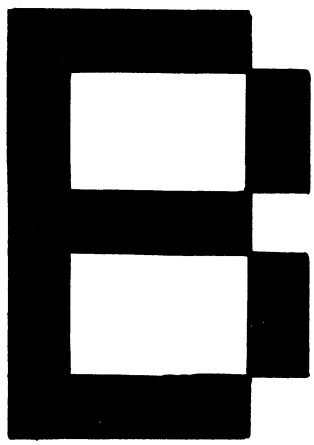
```
HLIN 10,13 AT 5
HLIN 10,13 AT 8
HLIN 10,13 AT 11
VLIN 5,11 AT 10
VLIN 6,7 AT 14
VLIN 8,9 AT 14
```

Looking at these two examples, it is not at all obvious how to use simple variable names to allow this repositioning to be handled without extra coding. Even if we did determine what variables to use, we would need a separate assignment statement for *each* such variable whenever we wished to plot the figure at a new location.

The solution is to use expressions instead of simple variables. To see how this should work, first consider the following version of the second set of graphics statements:

```
HLIN 10+0,10+3 AT 5+0
HLIN 10+0,10+3 AT 5+3
```

FIGURE 3.2 Block Letter B



```
HLIN 10+0,10+3 AT 5+6
VLIN 5+0,5+6 AT 10+0
VLIN 5+1,5+2 AT 10+4
VLIN 5+4,5+5 AT 10+4
```

Look at all those 10's and 5's! It is obvious that the second set of statements is derived from the first by adding 10 to all the numbers which represent a column and adding 5 to all the numbers which represent a row. This same idea may be applied systematically to move the block letter to any place on the screen. Now, if we use two variables, ROW and COL, to represent the values being added, we arrive at the general solution:

```
HLIN COL,COL+3 AT ROW
HLIN COL,COL+3 AT ROW+3
HLIN COL,COL+3 AT ROW+6
VLIN ROW,ROW+6 AT COL
VLIN ROW+1,ROW+2 AT COL+4
VLIN ROW+4,ROW+5 AT COL+4
```

If we place these statements into a subroutine, then calling that subroutine will cause a block letter B to be drawn with its upper left-hand corner located at the current ROW and COL location. The only requirement is that ROW and COL be set properly before the subroutine is called.

The statements may be thought of as applying to a "screen within a screen." The smaller screen is a graphics analog to a text window. The upper left-hand corner of this screen is represented by the pair (ROW, COL) and may be referred to as the BASE. The numbers added to ROW and COL in the various HLIN, VLIN, and PLOT statements are *relative* to the (ROW, COL) base. These are sometimes referred to as *offsets*, thus the title of this section. The concept is illustrated in Figure 3.3.

Giant Letters Screen

The program of Listing 3.2 presents a Giant Letters Screen. It plots whatever is keyed by the user in block-letter form on the low-resolution screen. The letters all fit into a 5 column by 7 row graphics matrix. The matrix may be thought of as a graphics window or subscreen.

Allowing an extra column for horizontal separation and an extra row for vertical separation between each individual character, the screen may be divided into 8 row by 6 column chunks. Given that the full 48 row by 40 column low-resolution screen will be used, this provides enough space for six rows of five giant letter characters.

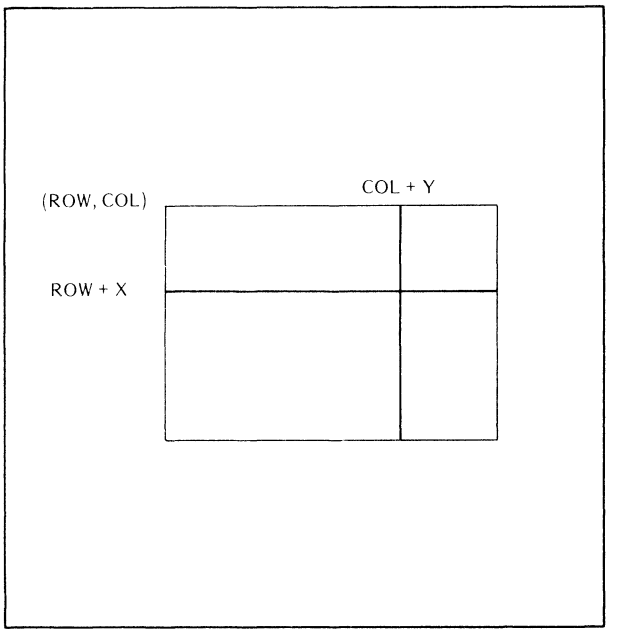


FIGURE 3.3 Base–Offset Graphics

Full-screen, low-res graphics

The mode in which information is displayed on the APPLE II screen is determined by a number of so-called "soft switches." These are dedicated addresses which do not hold *data*, but which when accessed (via PEEK or POKE) will cause a change of state to occur within the APPLE hardware configuration.

The screen switches actually occur in pairs. Each pair enables selection between two styles of screen operation. For example,

−16302	Selects full-screen mode; i.e., all text or all graphics
−16301	Selects mixed text and graphics

The behavior of these two switches interacts with the setting of a number of other switches. The two BASIC statements:

```
GR
TEXT
```

influence this interaction. In particular, GR selects low-resolution graphics in the mixed text and graphics mode. In this mode, the bottom of the screen displays four lines of text (which also becomes the scrolling window as well—see Chapter 2), and the rest is a 40 × 40 low-resolution graphics display.

After a GR statement has been executed, full-screen graphics may be selected by issuing the statement:

POKE −16302,0 or **POKE FULL, 0**
assuming **FULL=−16302**

There is a slight difficulty here in that what displays as a blank on the text screen does not display as black in low-resolution graphics. Thus, after selecting full-screen mode, if a black screen is desired the bottom 8 graphics rows must be "cleared" as follows:

```
COLOR = 0
FOR I=40 TO 47
HLIN 0,39 AT I
NEXT I
```

Combining these gives the standard subroutine SETFULL, which appears starting at line number 1000 in the Giant Letters Screen program.

Scrolling the low-res screen

The Giant Letters Screen program performs scrolling of the low-res graphics screen. The text window scrolling routine is part of the ROM Monitor and is normally limited to its prescribed function: scrolling parts of the text screen. When a GR statement is executed, the window parameters are set so that only the bottom four text lines will be scrolled. To get around this, we may explicitly set the window parameters back to scroll the entire screen.

In Giant Letters Screen, this is accomplished by the FULLWINDOW subroutine located at line 1050. Refer back to Chapter 2 for details. The SCROLL subroutine (=1100) works as follows.

Each time the ROM Monitor scrolling routine is called (CALL−912), the entire display is rolled up one *text* line. Since there are only 24 text lines but 48 graphics rows, each text line corresponds to two graphics rows. The bottom text line corresponds to graphics rows with numbers 46 and 47. The statements

```
HLIN 0,39 AT 46
HLIN 0,39 AT 47
```

serve to "erase" each new pair of graphics rows as they appear. The reason that this is necessary is that the ROM routine at −912 thinks it is scrolling text. Therefore, it fills the bottom text line with text blanks. As we mentioned above, these display in color when interpreted as low-res graphics.

Use one subroutine for each graphic element

The Giant Letters Screen implements the drawing of individual letters by the subroutines beginning at line number 5000. Each subroutine is responsible for drawing a single

character whose upper left-hand corner is located by the current values of ROW and COL. This approach gives maximum flexibility when redrawing figures at different places on the screen.

3. THE LOW-RES STRING INTERPRETER

One of the difficulties of using the low-resolution graphics features to any great extent is the large number of program statements which must be written. The Giant Letters Screen program is a good example. In order to be able to respond to any possible character that might be typed at the APPLE keyboard, a total of 96 different subroutines of several lines each must be written (see the Explorations section for more detail). In this section, we shall explore an interpreter which may be used to create quite complex graphics scenes with much more compact coding.

Compact Graphics Coding

Think about abbreviations. Suppose instead of writing:

```
HLIN COL+3,COL+7 AT ROW+5
VLIN ROW+1,ROW+4 AT COL
```

we wrote:

```
H375V140
```

The meaning of the latter is still perfectly clear to us, but unfortunately not to Integer BASIC. Writing:

```
10 H375V140
```

would only give us a beep and a:

```
*** SYNTAX ERR
```

for our efforts. On the other hand, writing:

```
10 FM$ = "H375V140"
```

is legal BASIC. It might be used to pass the "encoded" representation of the two graphics statements above to a BASIC program. The program would need a means of decoding or *interpreting* the string in order to produce the equivalent graphics commands.

This capability is provided by a collection of subroutines in the demonstration program of Section 3.4. Let's examine how it works.

When subroutine DECODE (=1000) is entered, it is assumed that the string variable FM$ contains a "format" string to be interpreted or decoded. The decoding is done on a command by command basis. Each individual command is represented by a single letter. The commands implemented are as follows:

Hxyz	Draws a horizontal line: **HLIN COL+x,COL+y AT ROW+z**
Vxyz	Draws a vertical line: **VLIN ROW+x,ROW+y AT COL+z**
Pxy	Plots a point: **PLOT COL+x,ROW+y**
Cx	Sets color: **COLOR = x**
Txy	Translates origin: **ROW=ROW+x** **COL=COL+y**
Dxyz	Draws a diagonal line connecting **ROW+x,COL+y and** **ROW+x+z,COL+y+z**
Bxyzw	Draws a solid block with boundaries: **ROW+x to ROW+x+z** **COL+y to COL+y+w**
Sxy	Slides origin back: **ROW=ROW−x** **COL=COL−y**
F	Fills the screen with the current color

Each command is carried by its own subroutine. These subroutines are responsible for setting up values needed by the command. This is done by calling the subroutine GETARG which translates characters in the format string into numeric values. This subroutine allows letters to be interpreted as well as digits. Letter A is assigned the value 10, B value 11, etc., until letter Z is assigned value 35. With this in mind, it is easy to understand the subroutine HRIZONTAL:

```
1155 GOSUB GETARG: RX=ARG
1160 GOSUB GETARG: CX=ARG
1165 GOSUB GETARG: AX=ARG
1170 HLIN COL+RX,COL+CX AT ROW+AX
1199 RETURN
```

The structure of the complete set of routines is shown in Figure 3.4.

The use of format strings may be thought of as a minigraphics language added onto BASIC. The routines just described form an interpreter for this language itself written in BASIC.

The advantage of the format strings is their compactness. The big disadvantage is their speed, or rather lack of it. You will no doubt agree when you try the demo program in the next section. Why even bother you might ask? The ideas represented by the format string interpreter may later be applied to provide similar capabilities, but with acceptable speed, by using assembler programming.

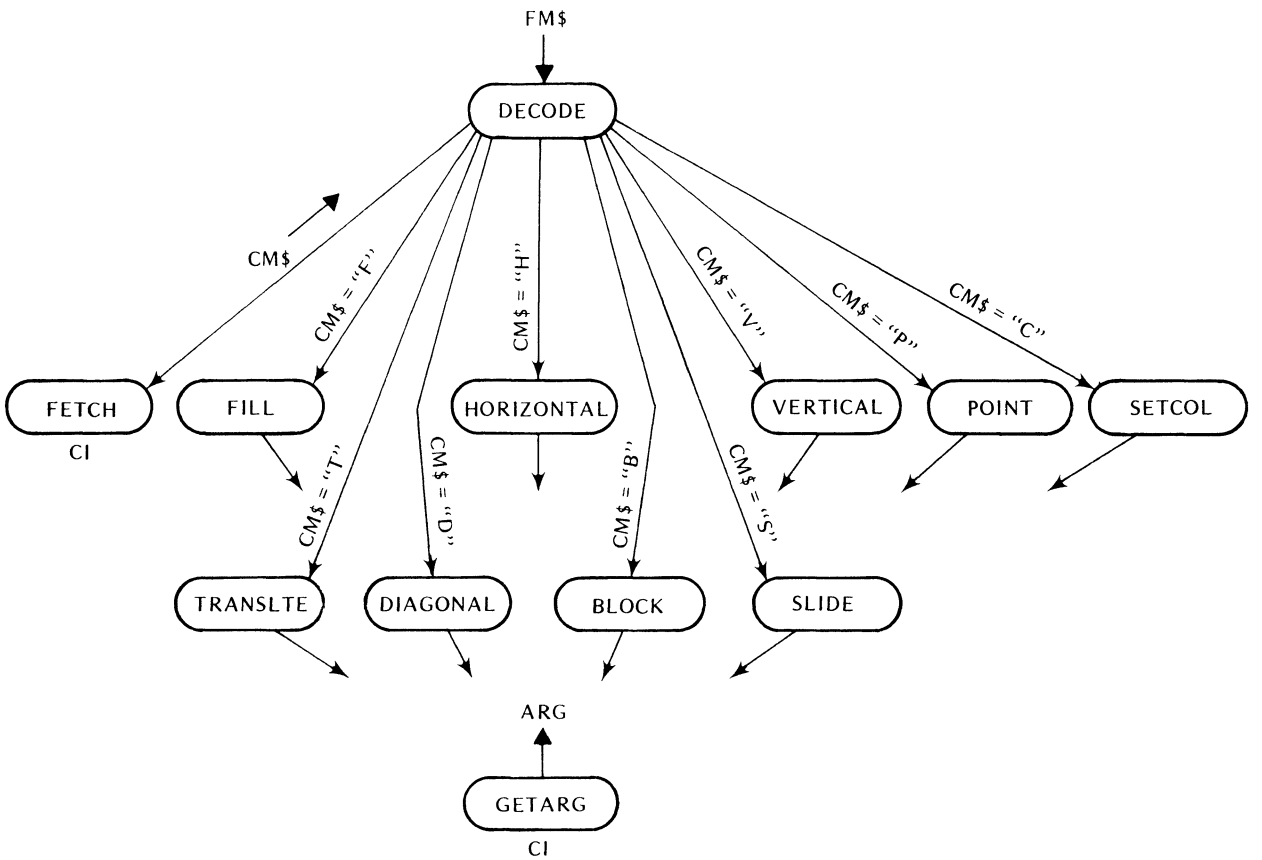


FIGURE 3.4 Structure of the Format String Decoder

4. APPLYING THE FORMAT STRING INTERPRETER

The program of Listing 3.3 provides a choice of four separate graphics demos. Two of the demos are left out on purpose to give you a chance to implement them yourself.

5. EXPLORATIONS

- Add subroutines to the Giant Letters Screen program to supply the letters, digits, and punctuation characters not already supported.
- Develop techniques using graph paper to lay out graphics figures and translate them into format strings.
- Modify the Giant Letters Screen program to allow user-defined characters to appear on the screen whenever a control character is input.
- Add to the Video program to allow more control over the sequence of choices between TICTAC, QUADTAC, and OCTTAC.

LISTINGS

LISTING 3.1 VIDEO TEST PATTERNS

```
>LIST
    1 REM ============================
    2 REM =                          =
    3 REM =   VIDEO TEST PATTERN     =
    4 REM =                          =
    5 REM =DR. RICHARD C. VILE, JR.=
    6 REM = ALL COMMERCIAL RIGHTS    =
    7 REM =       RESERVED           =
    8 REM =                          =
    9 REM ============================
   15 KBD=-16384:CLR=-16368:HOME=
      -936:BELL=-98
   20 TICTAC=1000:QUADTAC=1100
   21 POLICY=1200:WAIT=1300:OCTTAC=
      1400
   22 ESCAPE=128+27
  100 REM ======================
  101 REM = M A I N   L I N E =
  102 REM ======================
  110 GR
  115 IF RND (2) THEN 130
  120 R1= PDL (1)/25+2
  125 FOR I=1 TO RND (R1): GOSUB
      TICTAC: NEXT I
  129 GOTO 115
  130 IF RND (2) THEN 160
  135 R2= PDL (1)/25+2
  140 FOR I=1 TO RND (R2)
  145 GOSUB QUADTAC
  150 NEXT I
  155 GOTO 115
  160 REM
  165 R3= PDL (1)/25+2
  170 FOR I=1 TO RND (R3)
  175 GOSUB OCTTAC
  180 NEXT I
  185 GOTO 115
 1000 REM ========================
 1001 REM =  TIC-TAC SUBROUTINE =
 1002 REM ========================
 1010 FOR J=0 TO 39
 1015 GOSUB POLICY
 1020 HLIN 0,39 AT J: HLIN 0,39 AT
      39-J
 1030 VLIN 0,39 AT J: VLIN 0,39 AT
      39-J
 1035 GOSUB WAIT
 1040 NEXT J
 1049 RETURN
 1100 REM ============================
 1101 REM =  QUAD-TAC SUBROUTINE  =
 1102 REM ============================
 1110 FOR J=0 TO 19
 1115 GOSUB POLICY
 1120 HLIN 0,39 AT J: HLIN 0,39 AT
      19-J
 1125 VLIN 0,39 AT 20+J: VLIN 0,39
      AT 39-J
 1130 VLIN 0,39 AT J: VLIN 0,39 AT
      19-J
 1135 HLIN 0,39 AT 20+J: HLIN 0,39
      AT 39-J
 1140 GOSUB WAIT
 1145 NEXT J
 1149 RETURN
 1200 REM ============================
 1201 REM =  COLOR POLICY ROUTINE  =
 1202 REM ============================
 1210 C1= PDL (1)/25+2
 1215 IF RND (C1) THEN COLOR= RND
      (16)
 1249 RETURN
 1300 REM ========================
 1301 REM =   WAIT SUBROUTINE   =
 1302 REM ========================
 1310 KEY= PEEK (KBD): IF KEY<128
      THEN RETURN
 1315 POKE CLR,0
 1320 KEY= PEEK (KBD): IF KEY<128
      THEN 1320
 1325 POKE CLR,0
 1330 IF KEY#ESCAPE THEN RETURN
 1335 TEXT : CALL HOME
 1349 END
 1400 REM ========================
 1401 REM =  OCT-TAC SUBROUTINE  =
 1402 REM ========================
 1410 FOR J=0 TO 9
 1415 GOSUB POLICY
 1420 HLIN 0,39 AT J: HLIN 0,39 AT
      39-J
 1425 VLIN 0,39 AT J: VLIN 0,39 AT
      39-J
 1430 HLIN 0,39 AT 10+J: HLIN 0,39
      AT 29-J
 1435 VLIN 0,39 AT 10+J: VLIN 0,39
      AT 29-J
 1440 HLIN 0,39 AT 20+J: HLIN 0,39
      AT 19-J
 1445 VLIN 0,39 AT 20+J: VLIN 0,39
      AT 19-J
 1450 HLIN 0,39 AT 30+J: HLIN 0,39
      AT 9-J
 1455 VLIN 0,39 AT 30+J: VLIN 0,39
      AT 9-J
 1460 GOSUB WAIT
 1465 NEXT J
 1499 RETURN
```

LISTING 3.2 GIANT LETTERS SCREEN

```
>LIST
    1 REM ============================
    2 REM =                            =
    3 REM =  GIANT LETTERS SCREEN      =
    4 REM =                            =
    5 REM =DR. RICHARD C. VILE, JR.=
    6 REM = ALL COMMERCIAL RIGHTS      =
    7 REM =        RESERVED            =
    8 REM =                            =
    9 REM ============================
   15 KBD=-16384:CLR=-16368:HOME=
      -936:BELL=-98
   16 CLREOL=-868:FULL=-16302
   19 INIT=10000
   20 SETFULL=1000:FULLWINDOW=1050
      :SCROLL=1100
   21 GETKEY=1150:DRIVER=600
  200 REM ==================
  201 REM =  MAIN PROGRAM   =
  202 REM ==================
  205 GR : PRINT : PRINT : PRINT

  208 ROW=0:COL=0
  210 GOSUB SETFULL
  212 GOSUB FULLWINDOW
  215 CC= RND (16): IF CC=0 THEN
      215
  216 COLOR=CC
  218 GOSUB DRIVER
  220 COLOR=CC:C1=C1-32
  222 IF C1<0 THEN 270
  225 CHR=5000+25*C1
  250 GOSUB CHR
  270 COL=COL+6: IF COL<=35 THEN
      215
  272 COL=0:ROW=ROW+8
  274 IF ROW<=40 THEN 215
  275 GOSUB SCROLL
  280 GOTO 215
  600 REM ====================
  601 REM =  KEYBOARD DRIVER  =
  602 REM ====================
  605 GOSUB GETKEY
  610 C1=KEY-128
  612 IF C1#27 THEN 618
  614 ROW=0:COL=0
  615 COLOR=0: FOR J=0 TO 39: VLIN
      0,47 AT J: NEXT J: COLOR=CC
  616 GOTO 600
  618 IF C1=13 THEN 685
  619 IF C1=7 THEN 675
  620 IF (C1#8 AND C1#21) THEN RETURN
  622 IF C1#21 THEN 625
  623 C1=32: RETURN
  625 COL=COL-6: IF COL>=0 THEN 650

  630 COL=30:ROW=ROW-8: IF ROW>=0
       THEN 650
  635 COL=0:ROW=0
  650 COLOR=0
  655 FOR J=0 TO 6
  660 HLIN COL,COL+4 AT ROW+J
  665 NEXT J
  670 COLOR=CC: GOTO 600
  675 PLOT COL,ROW+5: PLOT COL+4,
      ROW+5
  677 VLIN ROW+1,ROW+5 AT COL+1
  679 VLIN ROW+1,ROW+5 AT COL+3
  680 VLIN ROW,ROW+6 AT COL+2
  681 PRINT "";: RETURN
  685 ROW=ROW+8: IF ROW>=40 THEN
      GOSUB SCROLL
  687 COL=0: GOTO 600
 1000 REM ============================

 1001 REM = SET FULL SCREEN GRAPHICS =

 1002 REM ============================

 1005 POKE FULL,0
 1010 COLOR=0
 1015 FOR I=40 TO 47: HLIN 0,39 AT
      I: NEXT I
 1049 RETURN
 1050 REM ==========================
 1051 REM =     FULLWINDOW           =
 1052 REM =                          =
 1053 REM =  SET GRAPHICS WINDOW TO=
 1054 REM = FULL SCREEN TO ALLOW    =
 1055 REM = SCROLLING LOW-RES.      =
 1056 REM ==========================
 1060 POKE 32,0: POKE 33,40
 1061 POKE 34,0: POKE 35,24
 1099 RETURN
 1100 REM ==========================
 1101 REM =       SCROLL             =
 1102 REM ==========================
 1105 SAVCOL= PEEK (48): COLOR=0
 1110 FOR J=1 TO 4
 1115 CALL -912
 1120 HLIN 0,39 AT 46: HLIN 0,39 AT
      47
 1125 NEXT J
 1130 ROW=40:COL=0
 1135 COLOR=SAVCOL
```

LISTING 3.2 *(cont.)*

```
1149 RETURN
1150 REM ==================
1151 REM =      GETKEY      =
1152 REM ==================
1155 KEY= PEEK (KBD): IF KEY<128
      THEN 1155
1160 POKE CLR,0
1199 RETURN
5001 REM =      SPACE       =
5002 REM ==================
5003 REM ==================
5024 RETURN
5025 REM ==================
5026 REM =  EXCLAMATION   =
5027 REM ==================
5030 VLIN ROW,ROW+4 AT COL+3
5032 PLOT COL+3,ROW+6
5049 RETURN
5050 REM ==================
5051 REM = DOUBLE QUOTE =
5052 REM ==================
5060 VLIN ROW,ROW+1 AT COL+1
5065 VLIN ROW,ROW+1 AT COL+3
5074 RETURN
5075 REM ==================
5076 REM =  POUND SIGN   =
5077 REM ==================
5080 VLIN ROW+1,ROW+5 AT COL+1
5082 VLIN ROW+1,ROW+5 AT COL+3
5085 HLIN COL,COL+4 AT ROW+2
5087 HLIN COL,COL+4 AT ROW+4
5099 RETURN
5400 REM ==================
5401 REM =      ZERO        =
5402 REM ==================
5405 HLIN COL+1,COL+3 AT ROW
5410 HLIN COL+1,COL+3 AT ROW+6
5415 VLIN ROW+1,ROW+5 AT COL
5420 VLIN ROW+1,ROW+5 AT COL+4
5424 RETURN
5825 REM ================
5826 REM =  LETTER A   =
5827 REM ================
5830 HLIN COL+1,COL+3 AT ROW
5835 VLIN ROW+1,ROW+6 AT COL
5837 VLIN ROW+1,ROW+6 AT COL+4
5840 HLIN COL,COL+4 AT ROW+3
5849 RETURN
5850 REM ================
5851 REM =  LETTER B   =
5852 REM ================
5855 VLIN ROW,ROW+6 AT COL
5857 HLIN COL,COL+3 AT ROW
5860 HLIN COL,COL+3 AT ROW+6
```

```
5865 HLIN COL,COL+3 AT ROW+3
5870 VLIN ROW+1,ROW+2 AT COL+4
5872 VLIN ROW+4,ROW+5 AT COL+4
5874 RETURN
5875 REM ==============
5876 REM =  LETTER C  =
5877 REM ==============
5880 VLIN ROW+1,ROW+5 AT COL
5882 HLIN COL+1,COL+3 AT ROW
5884 HLIN COL+1,COL+3 AT ROW+6
5886 PLOT COL+4,ROW+1
5888 PLOT COL+4,ROW+5
5899 RETURN
5900 REM ==============
5901 REM =  LETTER D  =
5902 REM ==============
5905 VLIN ROW,ROW+6 AT COL
5910 HLIN COL,COL+3 AT ROW
5915 HLIN COL,COL+3 AT ROW+6
5920 VLIN ROW+1,ROW+5 AT COL+4
5924 RETURN
5925 REM ==============
5926 REM =  LETTER E  =
5927 REM ==============
5930 VLIN ROW,ROW+6 AT COL
5932 HLIN COL,COL+4 AT ROW
5934 HLIN COL,COL+4 AT ROW+6
5940 HLIN COL,COL+3 AT ROW+3
5949 RETURN
5950 REM ==============
5951 REM =  LETTER F  =
5952 REM ==============
5955 VLIN ROW,ROW+6 AT COL
5960 HLIN COL,COL+4 AT ROW
5965 HLIN COL,COL+3 AT ROW+3
5974 RETURN
5975 REM ==============
5976 REM =  LETTER G  =
5977 REM ==============
5980 VLIN ROW+1,ROW+5 AT COL
5982 HLIN COL+1,COL+3 AT ROW
5984 HLIN COL+1,COL+3 AT ROW+6
5986 HLIN COL+3,COL+4 AT ROW+3
5988 VLIN ROW+4,ROW+5 AT COL+4
5990 PLOT COL+2,ROW+4
5999 RETURN
6000 REM ==============
6001 REM =  LETTER H  =
6002 REM ==============
6005 VLIN ROW,ROW+6 AT COL
6010 VLIN ROW,ROW+6 AT COL+4
6015 HLIN COL,COL+4 AT ROW+3
6024 RETURN
6025 REM ==============
```

LISTING 3.2 (*cont.*)

```
6026 REM =  LETTER I  =
6027 REM ==============
6030 VLIN ROW,ROW+6 AT COL+2
6035 HLIN COL+1,COL+3 AT ROW
6037 HLIN COL+1,COL+3 AT ROW+6
6049 RETURN
6050 REM ==============
6051 REM =  LETTER J  =
6052 REM ==============
6055 VLIN ROW,ROW+5 AT COL+3
6057 HLIN COL+1,COL+2 AT ROW+6
6058 PLOT COL,ROW+5
6074 RETURN
6075 REM ==============
6076 REM =  LETTER K  =
6077 REM ==============
6080 VLIN ROW,ROW+6 AT COL
6085 PLOT COL+1,ROW+3
6086 PLOT COL+2,ROW+2: PLOT COL+
     2,ROW+4
6087 PLOT COL+3,ROW+1: PLOT COL+
     3,ROW+5
6088 PLOT COL+4,ROW: PLOT COL+4,
     ROW+6
6099 RETURN
6100 REM ==============
6101 REM =  LETTER L  =
6102 REM ==============
6105 VLIN ROW,ROW+6 AT COL
6107 HLIN COL,COL+4 AT ROW+6
6124 RETURN
6125 REM ==============
6126 REM =  LETTER M  =
6127 REM ==============
6130 VLIN ROW,ROW+6 AT COL
6135 VLIN ROW,ROW+6 AT COL+4
6136 PLOT COL+1,ROW+1: PLOT COL+
     3,ROW+1
6137 PLOT COL+2,ROW+2
6149 RETURN
6450 REM ==============
6451 REM =  LETTER Z  =
6452 REM ==============
6455 HLIN COL,COL+4 AT ROW
6457 HLIN COL,COL+4 AT ROW+6
6460 PLOT COL+4,ROW+1
6462 PLOT COL,ROW+5
6465 PLOT COL+3,ROW+2
6467 PLOT COL+1,ROW+4
6470 PLOT COL+2,ROW+3
6474 RETURN
```

LISTING 3.3 FORMAT STRINGS DEMOS

```
*** SYNTAX ERR
>LIST
    1 REM ============================
    2 REM =                            =
    3 REM =  FORMAT STRINGS DEMOS      =
    4 REM =                            =
    5 REM =DR. RICHARD C. VILE, JR.=
    6 REM = ALL COMMERCIAL RIGHTS      =
    7 REM =        RESERVED            =
    8 REM =                            =
    9 REM ============================
   10 RX=CX=AX=BX
   15 KBD=-16384:CLR=-16368:HOME=
      -936:BELL=-98
   16 CLREOL=-868
   20 DECODE=1000:GET=1950:QUIT=5

   21 INTRO=20000:INIT=21000:GETARG=
      1050
   22 FETCH=1100:HRIZONTAL=1150:VERTIC
      AL=1200
   23 POINT=1250:SETCOL=1300:TRANSLTE=
      1350
   24 DIAGONAL=1400:BLOCK=1450:SLIDE=
      1500
   25 FILL=1550
   50 DIM A$(10),FM$(100)
   51 DIM DEMOS(10)
   90 GOSUB INIT
  100 REM ======================
  101 REM = M A I N   M E N U =
  102 REM ======================
  105 TEXT : CALL HOME: POKE 50,255
      : VTAB 5: TAB 5
  106 PRINT "CHOOSE ONE OF THE FOLLOWI
      NG"
  110 PRINT
  120 TAB 5: PRINT "(A) LARGE LETTERS
      SCREEN"
  125 TAB 5: PRINT "(B) SMALL LETTERS
      DEMO"
  130 TAB 5: PRINT "(C) ADDITION DRILL
      "
  135 TAB 5: PRINT "(D) SUBTRACTION DR
      ILL"
  140 TAB 5: PRINT "(E) EXIT"
  146 GOTO 150
  149 CALL BELL
  150 VTAB 14: TAB 5: CALL CLREOL:
       GOSUB GET
  155 TYPE=KEY- ASC("@")
  157 IF (TYPE<1) OR (TYPE>5) THEN
      149
```

LISTING 3.3 (*cont.*)

```
159 IF TYPE#QUIT THEN 165
160 CALL HOME: END
165 IF TYPE<>1 THEN SU=1
170 GOSUB DEMOS(TYPE)
199 GOTO 100
200 REM ============================
201 REM = LARGE LETTERS DRIVER SUB =
202 REM ============================
205 GR : PRINT : PRINT : PRINT
208 ROW=0:COL=0
210 POKE -16302,0
212 COLOR=0: FOR J=40 TO 47: HLIN
    0,39 AT J: NEXT J
215 CC= RND (16): IF CC=0 THEN
    215
218 GOSUB 600
220 COLOR=CC
225 FO=30000+10*C1
250 GOSUB FO
260 GOSUB DECODE
265 IF SU THEN RETURN
270 COL=COL+6: IF COL<=35 THEN
    215
272 COL=0:ROW=ROW+8
274 IF ROW<=40 THEN 215
275 GOSUB 690
280 GOTO 215
600 X= PEEK (-16384): IF X<128 THEN
    600
605 POKE -16368,0
610 C1=X-128
612 IF C1#27 THEN 618
614 ROW=0:COL=0
615 COLOR=0: FOR J=0 TO 39: VLIN
    0,47 AT J: NEXT J: COLOR=CC
616 GOTO 600
618 IF C1=13 THEN 685
619 IF C1=7 THEN 675
620 IF (C1#8 AND C1#21) THEN RETURN
622 IF C1#21 THEN 625
623 C1=32: RETURN
625 COL=COL-6: IF COL>=0 THEN 650
630 COL=30:ROW=ROW-8: IF ROW>=0
     THEN 650
635 COL=0:ROW=0
650 COLOR=0
655 FOR J=0 TO 6
660 HLIN COL,COL+4 AT ROW+J
665 NEXT J
670 COLOR=CC: GOTO 600
675 PLOT COL,ROW+5: PLOT COL+4,
    ROW+5
677 VLIN ROW+1,ROW+5 AT COL+1
679 VLIN ROW+1,ROW+5 AT COL+3
680 VLIN ROW,ROW+6 AT COL+2
681 PRINT "";: RETURN
685 ROW=ROW+8: IF ROW>=40 THEN
    690
687 COL=0: GOTO 600
690 POKE 32,0: POKE 33,40: POKE
    34,0: POKE 35,24: COLOR=0
692 FOR J=1 TO 4: CALL -912: COLOR=
    0: HLIN 0,39 AT 46: HLIN 0,
    39 AT 47: NEXT J
694 ROW=40:COL=0: GOTO 600
1000 REM ======================
1001 REM = DECODE SUBROUTINE =
1002 REM ======================
1005 CI=1
1006 IF CI> LEN(FM$) THEN RETURN
1007 GOSUB FETCH
1015 IF CM$="H" THEN GOSUB HRIZONTAL
1016 IF CM$="V" THEN GOSUB VERTICAL
1017 IF CM$="P" THEN GOSUB POINT
1018 IF CM$="C" THEN GOSUB SETCOL
1019 IF CM$="T" THEN GOSUB TRANSLTE
1020 IF CM$="D" THEN GOSUB DIAGONAL
1021 IF CM$="B" THEN GOSUB BLOCK
1022 IF CM$="S" THEN GOSUB SLIDE
1023 IF CM$="F" THEN GOSUB FILL
1049 GOTO 1006
1050 REM ======================
1051 REM = GET NEXT ARGUMENT =
1052 REM = A. K. A.  GETARG  =
1053 REM ======================
1055 ARG= ASC(FM$(CI,CI))-176
1056 IF ARG>9 THEN ARG=ARG-7
1060 CI=CI+1
1099 RETURN
1100 REM ======================
1101 REM = FETCH NEXT COMMAND =
1102 REM ======================
1105 CM$=FM$(CI,CI)
1110 CI=CI+1
1149 RETURN
1150 REM ========================
1151 REM = PLOT HORIZONTAL LINE =
1152 REM ========================
1155 GOSUB GETARG:RX=ARG
1160 GOSUB GETARG:CX=ARG
1165 GOSUB GETARG:AX=ARG
```

```
1170 HLIN COL+RX,COL+CX AT ROW+AX
1199 RETURN
1200 REM ======================
1201 REM = PLOT VERTICAL LINE =
1202 REM ======================
1205 GOSUB GETARG:RX=ARG
1210 GOSUB GETARG:CX=ARG
1215 GOSUB GETARG:AX=ARG
1220 VLIN ROW+RX,ROW+CX AT COL+AX
1249 RETURN
1250 REM ================
1251 REM = PLOT A POINT =
1252 REM ================
1255 GOSUB GETARG:CX=ARG
1260 GOSUB GETARG:RX=ARG
1265 PLOT COL+CX,ROW+RX
1299 RETURN
1300 REM ==================
1301 REM =      SETCOL      =
1302 REM ==================
1305 GOSUB GETARG: COLOR=ARG
1349 RETURN
1350 REM ===============
1351 REM =  TRANSLATE  =
1352 REM ===============
1355 GOSUB GETARG:ROW=ROW+ARG
1360 GOSUB GETARG:COL=COL+ARG
1399 RETURN
1400 REM ==============
1401 REM =  DIAGONAL  =
1402 REM ==============
1405 GOSUB GETARG:RX=ARG
1406 GOSUB GETARG:CX=ARG
1407 GOSUB GETARG:AX=ARG
1410 FOR I=0 TO AX-1
1415 PLOT COL+CX+I,ROW+RX+I
1420 NEXT I
1449 RETURN
1450 REM ================
1451 REM =      BLOCK      =
1452 REM ================
1455 GOSUB GETARG:RX=ARG
1456 GOSUB GETARG:CX=ARG
1457 GOSUB GETARG:AX=ARG
1458 GOSUB GETARG:BX=ARG
1460 FOR I=ROW+RX TO ROW+RX+AX
1465 HLIN COL+CX,COL+CX+BX AT I
1470 NEXT I
1499 RETURN
1500 REM ================
1501 REM =      SLIDE      =
1502 REM ================
1505 GOSUB GETARG:ROW=ROW-ARG
1510 GOSUB GETARG:COL=COL-ARG
1549 RETURN
1550 REM ================
1551 REM =      FILL      =
1552 REM ================
1555 FOR I=0 TO 47
1560 HLIN 0,39 AT I
1565 NEXT I
1599 RETURN
1900 GR : PRINT : PRINT : PRINT

1905 ROW=0:COL=0
1910 GOSUB 215
1920 COL=COL+6: IF COL<=35 THEN
     1910
1925 COL=0:ROW=ROW+8
1930 IF ROW<=32 THEN 1910
1935 GOTO 1900
1950 REM ============================

1951 REM =  GETKEY FOR MENU ROUTINE =

1952 REM ============================

1955 KEY= PEEK (KBD): IF KEY<128
      THEN 1955
1960 POKE CLR,0
1999 RETURN
2000 REM ==========================
2001 REM = ADDITION DRILL PROGRAM =
2002 REM ==========================
2005 GR : PRINT : PRINT : PRINT

2008 CC= RND (16): IF CC=0 THEN
     2008: COLOR=CC
2010 X= RND (9)+1:Y= RND (9)+1
2020 ROW=7:COL=15
2025 C1=X+48: GOSUB 220
2030 C1=43:ROW=15:COL=7: GOSUB 220

2040 C1=Y+48:COL=15: GOSUB 220
2045 HLIN 13,22 AT 23
2055 ROW=25:COL=8
2060 TRIES=0
2065 INPUT "ANSWER",ANS
2070 C1=48+ANS/10: IF ANS<10 THEN
     C1=32
2075 GOSUB 220
2080 C1=48+(ANS MOD 10)
2085 COL=15
2090 GOSUB 220
2095 IF ANS#X+Y THEN 2200
2100 PRINT "GOOD!!"
2105 FOR J=1 TO 100: NEXT J
2110 GOTO 2000
```

LISTING 3.3 (*cont.*)

```
2200 FOR K=1 TO 100:Z= PEEK (-16336
     )- PEEK (-16336): NEXT K
2205 TRIES=TRIES+1
2206 IF TRIES>4 THEN 2000
2210 PRINT "WRONG-TRY AGAIN!!"
2215 FOR K=1 TO 200: NEXT K
2220 GOSUB 2500:COL=8: GOSUB 2500

2225 GOTO 2065
2500 COLOR=0
2510 FOR K=0 TO 6: HLIN COL,COL+
     4 AT ROW+K: NEXT K
2520 COLOR=CC
2525 RETURN
3000 REM ======================
3001 REM =  SUBTRACTION DRILL  =
3002 REM ======================
3999 RETURN
4000 REM ======================
4001 REM = SMALL LETTERS DEMO =
4002 REM ======================
4005 GR :ROW=0:COL=0
4010 CC= RND (16): IF CC=0 THEN
     4010
4015 COLOR=CC
4020 FOR J=0 TO 35
4025 IF J>9 THEN 4028:C1=48+J
4026 GOTO 4030
4028 C1=55+J
4030 GOSUB 4500
4035 COL=COL+4: IF COL<=36 THEN
     4100
4040 COL=0:ROW=ROW+6
4050 IF ROW<=42 THEN 4100
4060 ROW=0:COL=0
4100 NEXT J
4110 END
4500 F0=31500+C1*10
4510 GOSUB F0
4520 GOSUB DECODE
4530 RETURN
8000 FOR I=0 TO 31 STEP 2
8005 COLOR=I/2
8010 VLIN 0,39 AT I: VLIN 0,39 AT
     I+1
8020 NEXT I
9000 GR
9010 FOR I=0 TO 1600
9020 COLOR= RND (16)
9030 PLOT RND (39), RND (39)
9040 NEXT I
9050 POKE 32,0: POKE 33,39: POKE
     34,0: POKE 34,23
9060 FOR I=1 TO 4
9070 CALL -912: COLOR=0: HLIN 0,
     39 AT 38
 9075 HLIN 0,39 AT 39
 9080 NEXT I
 9090 END
21000 REM ====================
21001 REM = INITIALIZATIONS =
21002 REM ====================
21010 DEMOS(1)=200
21011 DEMOS(2)=4000
21012 DEMOS(3)=2000
21013 DEMOS(4)=3000
21099 RETURN
30010 FM$="P42H343H244H145H046"
30015 RETURN
30020 FM$="P20H131H042P23H134H045P26"

30025 RETURN
30040 FM$="H046H042H045P03P23P43P24"

30045 RETURN
30050 FM$="P00P20P40P11P31P02P22P42P13
      P33P04P24P44P15P35P06P26P46"

30055 RETURN
30060 FM$="V121V123V451V453"
30065 RETURN
30070 FM$=""
30075 RETURN
30080 FM$=""
30085 RETURN
30090 FM$="P21H132H043H134P25"
30095 RETURN
30100 FM$="B0022B0322B2022B2322B5022B5
      322"
30105 RETURN
30110 FM$="V062H043B0022"
30115 RETURN
30120 FM$="V010V012V014H042H044V560V56
      2V564"
30125 RETURN
30130 FM$="V062V063H010H013H016H340H34
      3H346"
30135 RETURN
30140 FM$="V061V063H010H013H016H340H34
      3H346"
30145 RETURN
30150 FM$="H040H043H046V060V062V064"

30155 RETURN
30160 FM$="P20H131H132H043H044H135H136
      "
30165 RETURN
30170 FM$="V062H043"
```

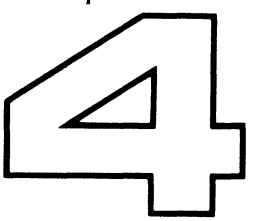
```
30175 RETURN
30180 FM$="H011H012H043H044P05P45"

30185 RETURN
30190 FM$="H042H043V450V454"
30195 RETURN
30200 FM$="V060V064V242P10P30P16P36"

30205 RETURN
30210 FM$="H010V011H121V121"
30215 RETURN
30220 FM$="H010V011H121V122H232V233H34
      3V344"
30225 RETURN
30230 FM$="H010H011H340H341H015H016H34
      5H346H132H133H134"
30235 RETURN
30240 FM$="H043P21P25"
30245 RETURN
30250 FM$="H130H131P22H043P24P15P35P06
      P46"
30255 RETURN
30260 FM$="V040V044V151V153V262"
30265 RETURN
30320 FM$=""
30325 RETURN
30330 FM$="V042P26"
30335 RETURN
30340 FM$="V011V013"
30345 RETURN
30350 FM$="H042H044V151V153"
30355 RETURN
30380 FM$="C1P00C2P01C3P02C4P03C5P04"

30385 RETURN
30390 FM$="B0222"
30420 FM$="V062P01P12P32P41P05P14P34P4
      5"
30425 RETURN
30430 FM$="H133V242"
30435 RETURN
30440 FM$="V562P16"
30445 RETURN
30450 FM$="H133"
30455 RETURN
30460 FM$="H125H126"
30465 RETURN
30470 FM$="P41P32P23P14P05"
30475 RETURN
30480 FM$="H136H130V154V150"
30485 RETURN
30490 FM$="H120P01V062H046"
30495 RETURN
30500 FM$="H130P01V124H133V460H046"
```

LISTING 3.3 (*cont.*)

```
30505 RETURN
30510 FM$="H130P01V124H233V454P05H136"

30515 RETURN
30520 FM$="V063H044P21P12P03"
30525 RETURN
30530 FM$="H040V030H033V454H136P05"

30535 RETURN
30540 FM$="H130P41V150H133H136V454"

30545 RETURN
30550 FM$="H040V124P33P24P15P06"
30555 RETURN
30560 FM$="H130H133H136V120V124V450V45
      4"
30565 RETURN
30570 FM$="H130H133H136V154V120P05"

30575 RETURN
30580 FM$="P22P24"
30585 RETURN
30590 FM$="P23V562P16"
30595 RETURN
30600 FM$="V460H460T11H260H171H022P42H
      682H093H034H494H015H355H795H016H
      786H137H577H268H359"
30605 RETURN
30610 FM$="H132H134"
30615 RETURN
30620 FM$="D123"
30625 RETURN
30630 FM$="H040H243P01V034P24P26"

30635 RETURN
30640 FM$="V060V034V564H040H046H243P22
      P45"
30645 RETURN
30650 FM$="P20P11P31V260V264H044"

30655 RETURN
30660 FM$="H030H033H036V060V124V454"

30665 RETURN
30670 FM$="H040H046V060"
30675 RETURN
30680 FM$="H030H036V060V154"
30685 RETURN
30690 FM$="H040H046V060H023"
30695 RETURN
30700 FM$="H040V060H023"
30705 RETURN
30710 FM$="H030H036V060V463H244"
```

```
30715 RETURN
30720 FM$="V060V064H043"
30725 RETURN
30730 FM$="H130H136V062"
30735 RETURN
30740 FM$="H140V063H136P05"
30745 RETURN
30750 FM$="V060P40P31P22P13P24P35P46"

30755 RETURN
30760 FM$="V060H046"
30765 RETURN
30770 FM$="V060V064P11P22P31"
30775 RETURN
30780 FM$="V060V064P12P23P34"
30785 RETURN
30790 FM$="V060V064H040H046"
30795 RETURN
30800 FM$="V060V034H040H043"
30805 RETURN
30810 FM$="H030H046V060V063"
30815 RETURN
30820 FM$="H030H033V060V124P24P35P46"

30825 RETURN
30830 FM$="H040H043H046V030V364P41P05"

30835 RETURN
30840 FM$="H040V062"
30845 RETURN
30850 FM$="V060V064H046"
30855 RETURN
30860 FM$="V020V024V341V343V562"
30865 RETURN
30870 FM$="V060V064V342P15P35"
30875 RETURN
30880 FM$="V010V014V560V564P12P32P23P1
      4P34"
30885 RETURN
30890 FM$="V010V014V362P12P32"
30895 RETURN
30900 FM$="H040H046P05P14P23P32P41"

30905 RETURN
30930 FM$="H230H236V063"
30935 RETURN
30940 FM$="P22P13P33P04P44"
30945 END

>
```

Chapter



Intelligent Programs

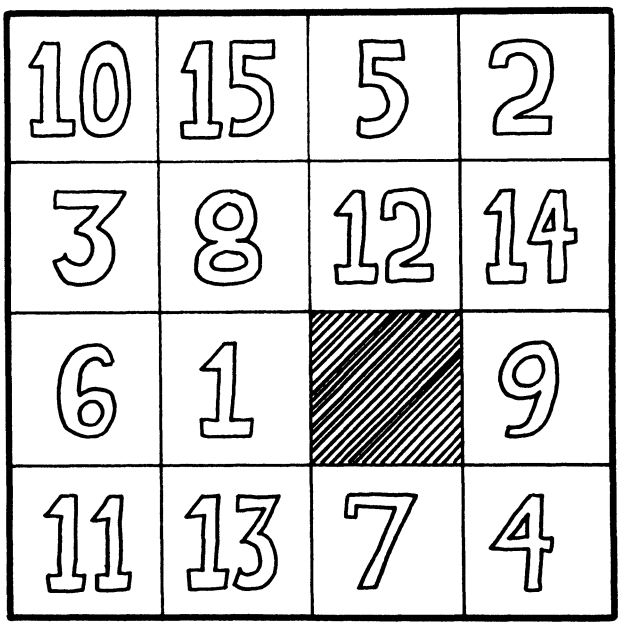
In this chapter, we present a program which performs an intelligent search. It will attempt to solve the famous Fifteen Puzzle. We begin by presenting a simpler program which presents a low-resolution graphics display of the Fifteen Puzzle and allows the user to solve the puzzle interactively.

1. THE FIFTEEN PUZZLE

The program of Listing 4.1 will apply some of the ideas of Chapter 3. It will also present an example of some fairly tricky techniques for representing information about puzzles and games. The Fifteen Puzzle program (see Listing 4.1) displays a low-resolution graphics representation of the puzzle itself. The user of the program directs the puzzle pieces to move around on the display in search of a solution. This is done by telling the program which piece to move next. The APPLE keyboard keys "R," "L," "U," and "D" are recognized by the program and represent the direction that the *moving* piece must take in order to move into the "hole" in the puzzle board. The letters, of course, stand for "right," "left," "up," and "down." This is illustrated by Figure 4.1.

In the position depicted, entering R would cause the 1 to move, entering L would cause the 9 to move, entering D would cause the 12 to move, and entering U would cause the 7 to move. Of course, in some positions there may be

FIGURE 4.1 Typical Fifteen Puzzle Position



many moves which are impossible. For example, if the hole is located in the lower right-hand corner of the puzzle board, then both U and L are impossible. Requesting either in such a position will cause the program to beep and ignore the request.

When the program is run, it scrambles the pieces of the puzzle and displays the result on the low-res screen. The player may then interact to his or her heart's content.

The first problem to consider in writing the Fifteen Puzzle program is that of representing the puzzle itself. The natural way to do this would be to use a *two-dimensional array:* PUZZLE(I,J). Unfortunately, Integer BASIC only allows one-dimensional arrays. This does not present an insurmountable obstacle. We may easily make do with a single dimensional array. We shall call the array which internally represents the puzzle: PUZZLE(16).

The 16 locations in the array correspond to the 16 puzzle locations, as indicated in the numbered diagram of Figure 4.2. The value stored in a given array entry is the number of the piece residing at the corresponding puzzle location. For example, in the puzzle situation depicted in Figure 4.1, we would have:

```
PUZZLE(1)  = 10
PUZZLE(2)  = 15
PUZZLE(3)  =  5
PUZZLE(4)  =  2
   . . .
PUZZLE(11) =  0
   . . .
PUZZLE(16) =  4
```

FIGURE 4.2 Correspondence Between Fifteen Puzzle Positions and BASIC Array Locations

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Notice that we have chosen to represent the presence of the hole by a value of 0.

Drawing the Puzzle

The puzzle is drawn on the low-res graphics screen, using the 40 × 40 part of the display. It is divided into 16 squares 10 × 10 in size, within which either one of the numbers 1 to 15 or the hole is drawn. The hole is represented by simply erasing that portion of the graphics screen. Each of the 16 puzzle slots has a base location in the sense of Chapter 3. These are shown in Figure 4.3.

In order to be able to redraw the contents of a given location, the program must be able to convert the location number (represented by the circled numbers in Figure 4.3) into its row and column base values. The subroutine CONVERT accomplishes this. The two statements:

```
ROW = (PLACE-1)/4*10
COL = ((PLACE-1)MOD 4)*10
```

provide the correct values. For example, PLACE = 10 yields

```
ROW = (10-1)/4 * 10 = (9/4)*10 = 2*10 = 20
COL = ((10-1)MOD 4)*10 = (9 MOD 4)*10 =
      1*10 = 10
```

The pictures of the numbers are drawn by individual subroutines in the style of Chapter 3. These routines are located beginning at line 8000 in both versions of the program given in this chapter.

FIGURE 4.3 Graphics Base Locations for Drawing Fifteen Puzzle Pieces

0,0 (1)	0,10 (2)	0,20 (3)	0,30 (4)
10,0 (5)	10,10 (6)	10,20 (7)	10,30 (8)
20,0 (9)	20,10 (10)	20,20 (11)	20,30 (12)
30,0 (13)	30,10 (14)	30,20 (15)	30,30 (16)

Moving the Pieces

Moves in the Fifteen Puzzle programs are indicated by the directions R, L, U, and D. In the interactive version, the player keys in these directions in order to solve the puzzle. In the intelligent version of the program, the program itself attempts to solve the puzzle. This means that it will calculate the moves to be made. The results of such calculations will still be given internally by directions in order to use the same program "machinery."

Now think about the problem of actually carrying out a move. Given a direction in which to move, the program must:

- Determine if the move is legal.
- Determine which piece is to be moved, by determining the location of the piece.
- Carry out the move in the internal representation of the puzzle.
- Update the display to reflect the move that was made internally.

Let us consider these points in more detail. Given a particular location of the *hole*, there are specific locations corresponding to each of the possible directions of movement. Some of these may be "off the board." Consider, for example, Figure 4.4. In this instance, there are no board locations corresponding to the directions D and R. The program must have this information at its disposal in order to reject these moves in such a situation.

The board connections are represented in the program by four additional arrays:

R(16)
L(16)
U(16)
D(16)

The contents of these arrays represent the board locations which would move into a given hole location via a given direction. For example, R(10) represents the board location from which a piece could move in the direction right into the hole located at board position 10. Sit and think about that for a few minutes, since it may be a bit tricky to understand at first.

Each of these arrays will contain some entries that correspond to impossible moves. The value stored in the arrays in those cases will be -1. This will enable the program to detect and reject impossible moves. The arrays are set up by the INIT subroutine starting at line 7140.

Before leaving this section, it must be pointed out that not all of the puzzle positions generated by the program are "solvable." In fact, exactly *half* of them are not. To understand what we mean by solvable consider Figure 4.5.

Starting with the position shown, it is mathematically impossible to restore the puzzle to the "correct" numerical order of 1 to 15. The position shown is said by mathematicians to have opposite *parity* from the standard solved position. There are two possible parities that a puzzle position may have—*even* and *odd*. Starting with a position of even parity and moving pieces legally (that is, without removing them from the puzzle tray and putting them back reordered), only positions of even parity may be reached. Likewise with positions of odd parity. Since

FIGURE 4.4 Impossible Moves Illustrated

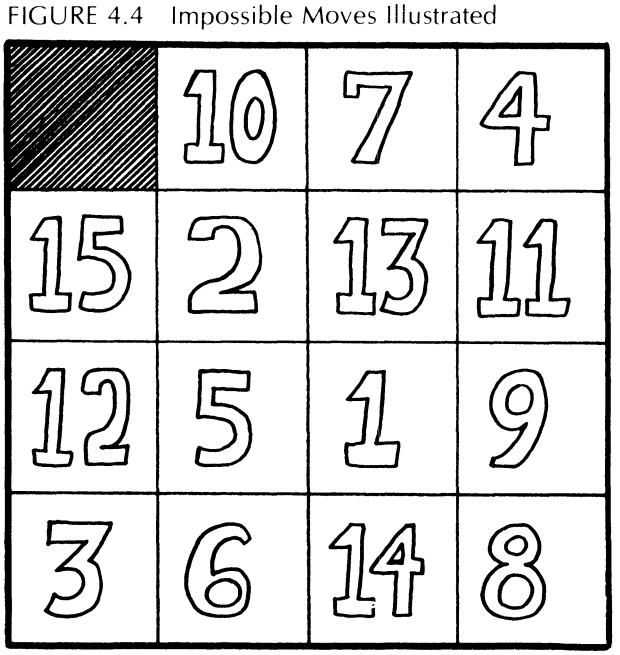
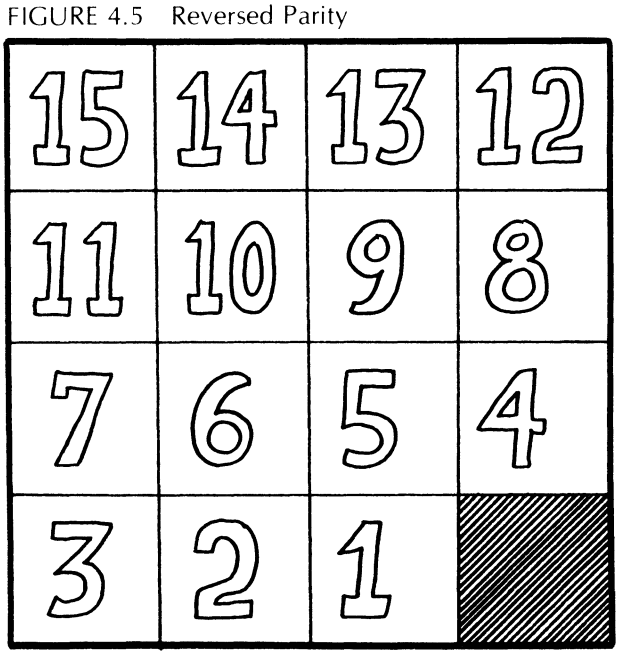


FIGURE 4.5 Reversed Parity



the position of Figure 4.5 has odd parity and the position of the "standard" solution has even parity, one cannot be reached from the other. This was the original basis for the Fifteen Puzzle. Its inventor, Sam Loyd, sold the puzzle arranged in the position of Figure 4.5 and offered a handsome prize to anyone who could demonstrate a way to restore it to the standard position. Of course, Sam's offer was a safe one, but it took quite some time before mathematicians exposed his trickery and many people spent ridiculous amounts of time trying to solve the unsolvable puzzle!

Since the Fifteen Puzzle program generates its board positions using the RND function, there is no guarantee that the result will be of even parity. Users of the program may find out what the parity of the position is by keying H, for help. A subroutine calculates the parity of the permutation and informs the user.

The subroutine OK is responsible both for making the move in the internal representation of the puzzle and for updating the display to reflect the result. The process consists of drawing the moving piece in the current hole location, then erasing the slot where the moving piece originated. The hole location then becomes the slot previously occupied by the moving piece. See lines 5110 to 5199 of Listing 4.1 for the details.

Figure 4.6 shows the hierarchy of the subroutines used in the Fifteen Puzzle program. As you can see, the program is quite simple in structure. The subroutines OK, CONVERT, ERASE, and the unnamed subroutines following ERASE form a pool of "expertise" which is invoked by the MOVE routines: MOVE, MOVR, MOVL, MOVUP, and MOVD.

2. AUTOMATING THE FIFTEEN PUZZLE

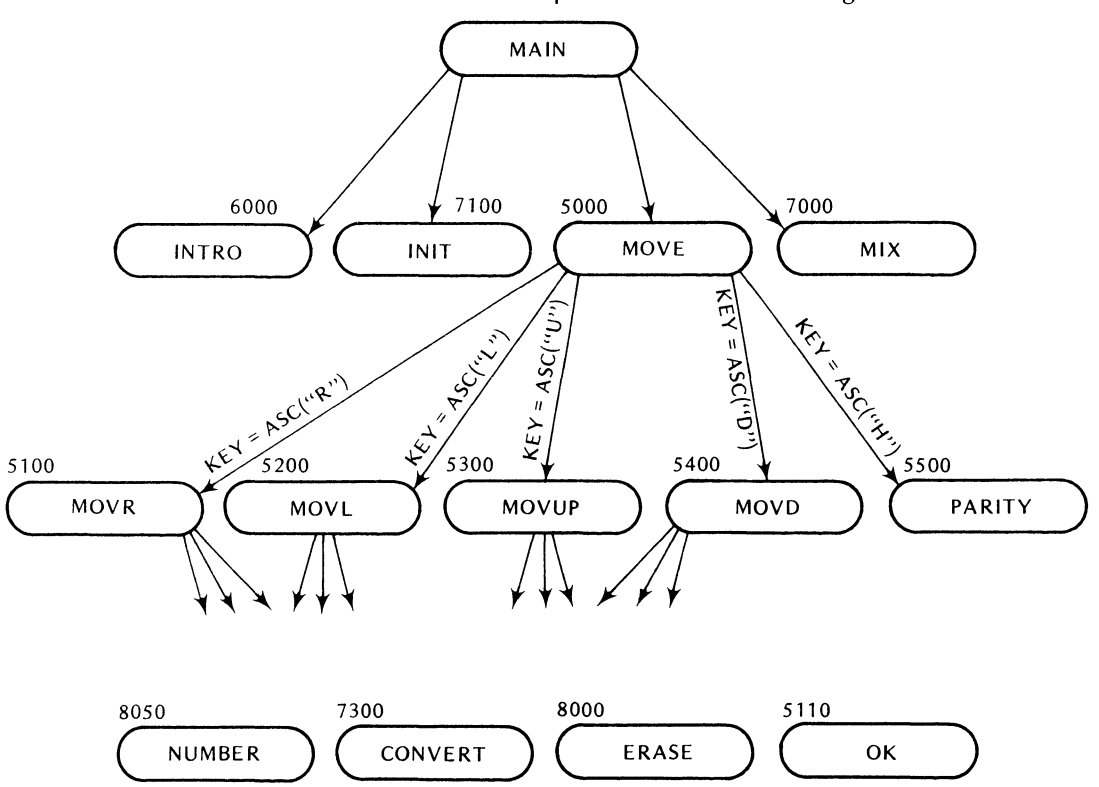
For the remainder of the chapter, we shall attempt to create a program that solves the Fifteen Puzzle automatically. This will lead us to invent ways to represent our *knowledge* of the Fifteen Puzzle and to use those representations to "drive" the puzzle-solving program.

To start with, we need to choose a *strategy*. Our approach will be *goal-directed,* with the ultimate goal being to solve the entire puzzle. In programming terms, we are seeking to make the following assertions true:

PUZZLE(I)=I for all I=1,2,3,...,15
PUZZLE(16)=0

In order to achieve this, we shall attempt to reach successive *subgoals*. The subgoals will focus on getting more and more of the puzzle into its correct order. The fundamental strategy will be:

FIGURE 4.6 Subroutine Call Graph for Fifteen Puzzle Program



- Determine the smallest value of I for which PUZZLE(I)#I.
- Move the Ith piece "closer" to its correct position.

The subgoal will be to get the "next" piece into correct order. This is, in fact, the way most human beings approach the problem. The trick in building up the program's "knowledge base" will be to identify how to represent "closer to" in programmatic terms.

Notation

Many times finding a solution to the problem depends on identifying an appropriate notation. In the case of the Fifteen Puzzle, we shall focus our attention on pertinent row and column values. In particular, our version of the search program will use the following locations and their row and column values:

Source	The board location of the piece which is next to be moved to its correct position.
Destination	The board location to which the moving piece is targeted. For PIECE=I, the destination location=I.
Hole	The board location where the hole is residing. This is most important, of course, since it determines the possible moves which may be made.

Program variables are:

SROW, SCOL	Row and column of Source
DROW, DCOL	Row and column of Destination
HROW, HCOL	Row and column of Hole

Situational Analysis

There are an astronomical number of possible positions of the Fifteen Puzzle. The number which may be reached from a given starting position is exactly half of all possible positions. Its value is:

$$1/2(16!) = 16 \times 15 \times 14 \times 13 \times 12 \times 11 \times 10 \times 9 \times 8 \times 7 \times 6 \times 5 \times 4 \times 3$$
$$= 10{,}461{,}394{,}944{,}000$$

or over ten trillion positions! That is far too many positions to store in a program, even on a supercomputer; much less on an APPLE. The key to reducing the problem to a manageable size is to recognize *patterns*. General conditions must be formulated in terms of the information introduced above and corresponding actions given which should be taken when those conditions are true. The conditions and the actions may then be incorporated into the SEARCH routine. We could attempt a complete analysis before any programming is done; however, we shall opt for a more scientific approach. That is, we shall first *theorize*, then experiment. The experiment will be carried out by embodying the theory in a program and then observing its behavior. When the program stumbles, we will locate a flaw in the theory. The theory may then be refined and the additional "knowledge" added to the program. These actions will be carried out in a loop, with the hope that eventually the theory will prove completely adequate.

3. A BASIC THEORY OF FIFTEEN PUZZLE SOLUTION

The program in Listing 4.2 presents an attempt to solve the Fifteen Puzzle via a computer program, in this case an Integer BASIC program. The operative word here is *attempt*, since we fall far short of the goal. Nonetheless the program provides an interesting and nontrivial example of a *procedural knowledge base*. The statements of the program contain general knowledge about the Fifteen Puzzle and when the program runs it may be described as *thinking about* the puzzle as it attempts to reach a solution. The knowledge in the program is the result of the author's analysis of the problem, but the reader is free to add to this knowledge base and teach the program more about the puzzle.

In this section, we shall explain *some* of the thought processes behind the SEARCH routine. We later invite the reader to dig out the rest of the theory from the program itself, either by analyzing the program code or by empirical analysis: observing the program's behavior.

Finding and Using Suitable Subgoals

We have already indicated in the last section that our approach will involve a "piece at a time" strategy. Thus, when presented with the situation of Figure 4.7, the program will attempt to put the 4 into its correct final position.

We shall refer to the piece being moved as the "*mover*." In the process of getting the mover to its correct position, we must get it to its correct row and to its correct column. It might seem that simply devising an algorithm to accomplish both those aims in a straightforward fashion would be adequate. If that were the case, we could close this chapter here and now and all go home. In fact, any simple general scheme will fail. It will have the effect of destroying past progress in many cases, since it does not

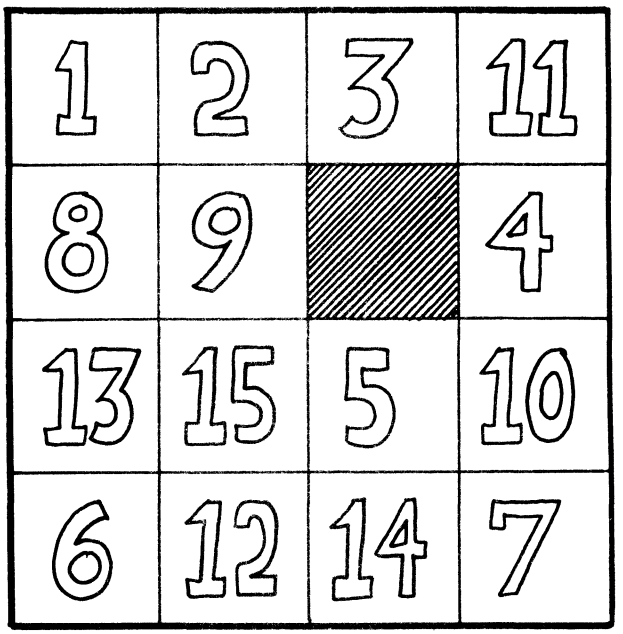


FIGURE 4.7 Concept of the "Mover"

take context into account and is not aware of the many special situations that may arise in the puzzle. Once a piece has been moved to its correct position, the SEARCH routine must not unknowingly move it, otherwise the program will easily catch itself in an infinite loop—we give an example below.

> A large part of the difficulty of automating the Fifteen Puzzle solution is the care required not to destroy or undo carefully wrought progress. This becomes more and more important as more and more of the puzzle pieces are brought to their correct positions.

Puzzle Analysis

The key to progress lies in identifying enough general situations and breaking them down into exhaustive cases so that from any possible situation moves will be made that make progress and do not lead the program into a cycle. To illustrate the difficulty, let us take an example of what can happen with an inadequate level of detail in the theory.

Suppose we adopt the strategy:

- Move the mover to its correct column.
- Move the mover to its correct row.

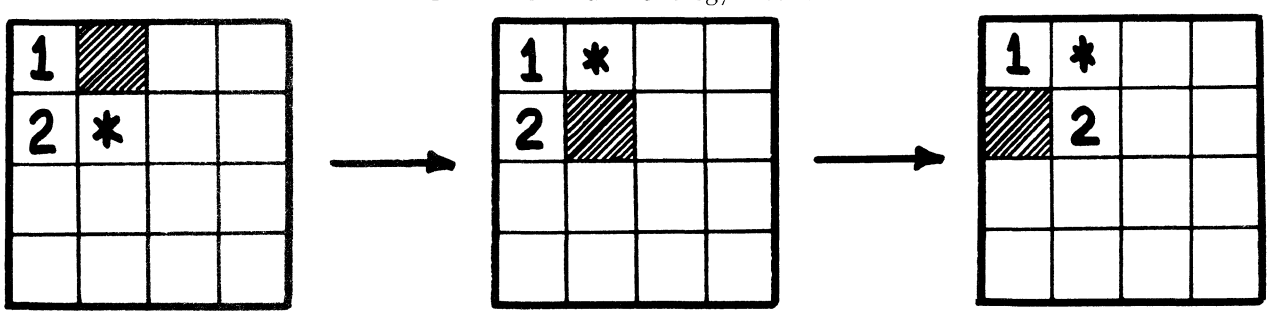
We proceed by moving the hole into positions that directly facilitate the two subgoals. For example, in getting the mover to its correct column, we place the hole adjacent to the mover and in the direction which the mover needs to go. Likewise in getting the mover to its correct row. In order to place the hole, we simply make the minimum number of moves required: we may make the hole follow a standard L-shaped path from its current location to the desired one. (This may in many cases simply be a straight line along a row or column.) Consider the sequence in Figure 4.8. The result of the sequence is to place the mover, 2, into its correct column position. Now we wish to move it to its correct row. This necessitates placing the hole above the 2. In all likelihood, the simple L-shaped move sequence will cause us to move the 1 down (in order to move the * left). But then 2 is no longer the mover! The program will cleverly realize this and move 1 back to its correct slot. Then the program will not so cleverly move the 1 down again for the same reason as before; then up, then down, then up, ...

The SEARCH routine will not be able to get very far without taking a much more detailed approach. It must consider the location of other pieces as well as that of the mover. It will not be able to take one simple, clear-cut course of action.

Case Analysis

We have not given up on the idea of subgoals. We must go more deeply into the possible puzzle situations and handle a potentially large number of *cases*. Each case represents a slightly different combination of circumstances which requires a different approach to solution than do other cases. Take as another example the situation shown in Figure 4.9.

FIGURE 4.8 Naive Strategy Illustrated



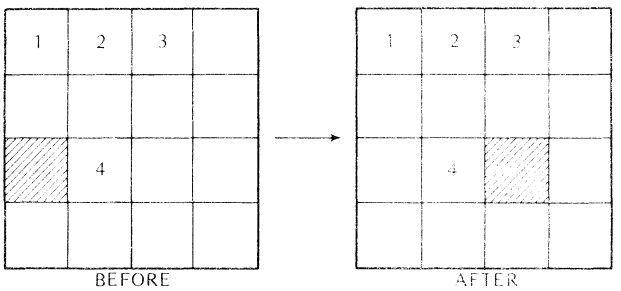


FIGURE 4.9 Achieving a Subgoal

In the situation of Figure 4.9, we may decide to position the hole to the right of the mover in order to get the mover closer to its destination column. A multiple move of ULLD will accomplish that goal. There is a danger here, however. If the mover is located on the bottom row of the puzzle, then the multiple move ULLD should not be attempted for obvious reasons. In that *case*, a multiple move of DLLU would be the appropriate action. In developing the program's knowledge base, many such situations will arise. The program will assist us in finding out our bungles. If an illegal move is attempted, it will beep continuously until we interrupt it. If we introduce a loop into the strategy, we will be able to see the loop being perpetrated on the screen.

A Partial Exegesis

We now describe in detail some of the inner workings of the Fifteen Puzzle SEARCH routine. The reader is invited to follow along in the listing of the program itself, and to figure out those parts of the strategy which we do not describe.

As indicated above, the analysis uses information about the source and destination locations of the mover and the location of the hole. The program makes decisions based on the values of the program variables which describe these locations.

The top level decision is based on a comparison of the source and destination *columns:* the subgoal is to make these equal. Thus, there are three possibilities:

1. DCOL > SCOL
2. DCOL < SCOL
3. DCOL = SCOL

In case 3, the subgoal has been achieved and further subgoals are then considered.

Representation of TRUE and FALSE

Since Integer BASIC represents TRUE by the numeric value 1 and FALSE by the numeric value 0, the following statement "works":

```
2075 GOTO (DCOL>SCOL)*2085 +
(DCOL<SCOL)*2080 + (DCOL=SCOL)*2090
```

This single GOTO statement replaces a complex series of IF tests and extra GOTO statements. Using individual IF's we would have to write something like the following:

```
2075 IF DCOL=SCOL THEN 2090
2076 IF DCOL>SCOL THEN 2085
2077 IF DCOL<SCOL THEN 2080
```

Of course, we could leave out the last statement and allow the program to fall into line 2080, but if later on, we wished to send it to some remote line number on condition DCOL<SCOL, the statement would have to be inserted.

Which of these two approaches is *better* is a matter of some dispute. The first is more elegant and takes less space. The second *may* be more comprehensible. Which approach *you* use is a matter of personal style, but it is probably a good idea to be consistent in your choice. In the Fifteen Puzzle SEARCH routine, the earlier approach has been used consistently.

Getting back to the analysis: if DCOL > SCOL, the next thing to look at is the relationship of the hole row and the mover row:

1. HROW = SROW
2. HROW > SROW
3. HROW < SROW

The object is to position the hole so that progress may be made toward the desired subgoal of DCOL = SCOL.

If case 1 holds:

HROW = SROW

and the hole and the mover are in the same row. We may attempt to place the hole on the side of the mover which allows it to get closer to its destination column. Recall that we have already assumed that the mover needs to go right to reach its destination, since DCOL > SCOL. Now we need to compare the hole column and the mover column in order to determine what to do with the hole.

HCOL < SCOL

This says that the hole is "behind" the mover. In order to get the mover further to the right, the hole must be positioned "beyond" the mover.

Since the hole and the mover are in the same row, this amounts to the two further cases shown in Figure 4.10. In the first case, a move of "L" is correct. In the second case, a multiple move is appropriate. ULLD will take the hole to the other side of the mover as shown in Figure 4.11. This won't work if the hole (and the mover) are in the third row. In that situation, the multiple move of DLLU should be used instead.

This should be enough to give you the flavor of the

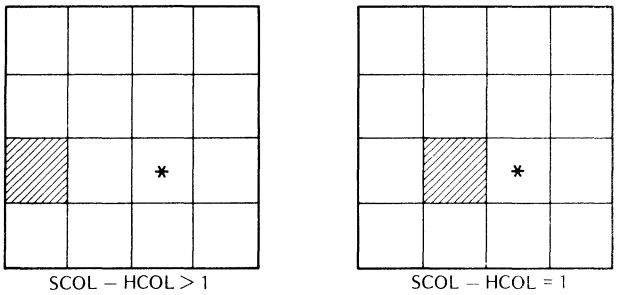


FIGURE 4.10 Case Analysis

theory. It is really necessary to sit down and reflect on the puzzle and various situations that may arise in order to understand it and improve on it. There are just too many special situations to consider.

Before ending this section, let us look at one more piece of the theory. In the situation depicted in Figure 4.7, a more complex multiple move is appropriate, and is usually the method adopted by a human "puzzler." Note some of the conditions that are necessary for this to be appropriate:

DCOL = SCOL
HROW = SROW
HCOL = 2

This situation is handled at line 2100 of the program. See if you can trace the program's arrival at that point.

Two ways to count

There are two ways of numbering items in *serial* or counting order. The first way is referred to as *1-indexing* and is the way most people count things: namely, starting at 1. The other method is referred to as *0-indexing* and is often more convenient in computer programs for one reason or another. In the Fifteen Puzzle SEARCH routine, both the row numbering and the column numbering have been done using 0-indexing. This is why the condition above states that HCOL = 2, instead of HCOL = 3.

4. EXPLORATIONS

- Modify the puzzle scrambling routine so that it always produces a permutation of even parity.
- Work out a complete description of the theory embodied by the SEARCH routine. See whether or not a more compact description is possible than the "wordy" approach we have taken in the text.
- Identify weaknesses in the SEARCH routine's theory. Attempt to fix at least one or two of the weaknesses. What problems will you encounter with the line numbering in the program?
- Think about alternate theories and strategies for creating a SEARCH routine. One possibility would be to have separate knowledge bases for each of the numbered pieces in the puzzle. Then, for example, the multiple move RRDLLLURDRRU discussed above might be specifically tied to pieces 4, 8, and 12.
- Explore the possibility of an interpretive approach to the SEARCH routine. Try inventing a notation which could be stored in the BASIC program which would represent knowledge about the puzzle and which the program could interpret in some fashion.
- What other popular puzzles or pastimes might it be possible to solve using a program? Develop graphic representations for at least one and attempt to automate it. (An obvious example might be a program that plays Tic-Tac-Toe against a human opponent.)

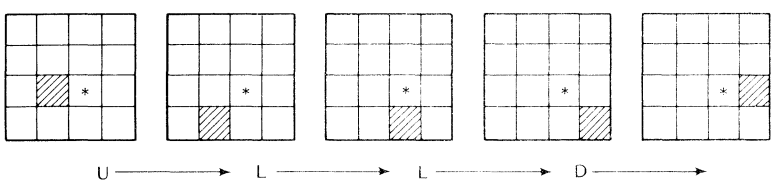


FIGURE 4.11 Multiple Move Strategy Illustrated

LISTINGS

LISTING 4.1 FIFTEEN PUZZLE

```
>LIST
    1 REM ============================
    2 REM =                          =
    3 REM =     FIFTEEN PUZZLE       =
    4 REM =                          =
    5 REM =DR. RICHARD C. VILE, JR.=
    6 REM = ALL COMMERCIAL RIGHTS  =
    7 REM =        RESERVED          =
    8 REM =                          =
    9 REM ============================
   10 KBD=-16384:CLR=-16368
   15 DIM PUZZLE(16),A$(10)
   16 DIM R(16),L(16),U(16),D(16)

   20 MIX=7000:INIT=7100:INTRO=6000

   21 MOVE=5000:ERASE=8000
   22 WAIT=7200:CONVERT=7300
   23 MOVR=5100:MOVL=5200:MOVUP=5300
      :MOVD=5400
   24 OK=5110:PARITY=5500
   25 POKE CLR,0
   50 GOSUB INTRO
   55 GOSUB MIX
   60 GOSUB INIT
   70 GOSUB MOVE
   75 IF FLAG=1 THEN 90
   80 GOTO 70
   90 FLAG=0
   92 PRINT "STARTING OVER"
   95 GOTO 55
 5000 REM ============================
 5001 REM =           MOVE           =
 5002 REM ============================
 5004 DONE=0
 5005 IF DONE THEN RETURN
 5008 GOSUB WAIT
 5010 IF KEY# ASC("R") THEN 5020
 5015 GOSUB MOVR: GOTO 5005
 5020 IF KEY# ASC("L") THEN 5030
 5025 GOSUB MOVL: GOTO 5005
 5030 IF KEY# ASC("U") THEN 5040
 5035 GOSUB MOVUP: GOTO 5005
 5040 IF KEY# ASC("D") THEN 5050
 5045 GOSUB MOVD: GOTO 5005
 5050 IF KEY# ASC("Q") THEN 5060
 5055 TEXT : CALL -936: END
 5060 IF KEY#155 THEN 5070
 5065 TEXT : CALL -936:FLAG=1: RETURN

 5070 IF KEY# ASC("H") THEN 5080
 5075 GOSUB PARITY: GOTO 5005
 5080 CALL -198: REM  BEEP
 5085 GOTO 5005
 5100 REM ============================
 5101 REM =          MOVR            =
 5102 REM = TRY TO MOVE RIGHT       =
 5103 REM = INTO THE HOLE.          =
 5104 REM ============================
 5105 SLOC=R(HOLE): IF SLOC#-1 THEN
      5110
 5108 CALL -198: RETURN
 5110 REM ============================
 5111 REM =           OK             =
 5112 REM = MOVE IS LEGAL - IT'S    =
 5113 REM = OK TO MAKE IT.          =
 5114 REM ============================
 5115 PLACE=HOLE
 5120 GOSUB CONVERT
 5125 SNUM=PUZZLE(SLOC)
 5130 GOSUB ERASE+SNUM*50
 5140 PLACE=SLOC
 5145 GOSUB CONVERT
 5160 REM  UPDATE HOLE INFO
 5165 PUZZLE(HOLE)=SNUM
 5170 PUZZLE(SLOC)=0
 5175 HOLE=SLOC
 5185 SNUM=0: GOSUB ERASE
 5190 DONE=1
 5199 RETURN
 5200 REM ============================
 5201 REM =          MOVL            =
 5202 REM = TRY TO MOVE LEFT        =
 5203 REM = INTO THE HOLE.          =
 5204 REM ============================
 5205 SLOC=L(HOLE): IF SLOC#-1 THEN
      5210
 5207 CALL -198: RETURN
 5210 GOSUB OK
 5299 RETURN
 5300 REM ============================
 5301 REM =          MOVUP           =
 5302 REM = TRY TO MOVE UP INTO     =
 5303 REM = THE HOLE.               =
 5304 REM ============================
 5305 SLOC=U(HOLE): IF SLOC#-1 THEN
      5310
 5307 CALL -198: RETURN
 5310 GOSUB OK
 5399 RETURN
 5400 REM ============================
 5401 REM =          MOVD            =
 5402 REM = TRY TO MOVE DOWN        =
 5403 REM = INTO THE HOLE.          =
 5404 REM ============================
 5405 SLOC=D(HOLE): IF SLOC#-1 THEN
```

```
    5410
5407 CALL -198: RETURN
5410 GOSUB OK
5499 RETURN
5500 REM  CALCULATE PARITY OF THE
5501 REM   PERMUTATION
5502 PRINT : PRINT "     HELP IS ON T
    HE WAY"
5505 SW=0
5510 FOR X=1 TO 15
5515 FOR Y=X+1 TO 16
5520 XX=PUZZLE(X):YY=PUZZLE(Y)
5525 IF XX=0 THEN XX=16: IF YY=0
     THEN YY=16
5530 IF XX>YY THEN SW=SW+1
5532 VTAB 24: TAB 10: PRINT "***"
    : VTAB 24: TAB 10: PRINT "   "
    :
5535 NEXT Y,X
5537 IF HOLE=2 OR HOLE=4 OR HOLE=
    5 OR HOLE=7 OR HOLE=10 OR HOLE=
    12 OR HOLE=13 OR HOLE=15 THEN
    5560
5540 IF (SW MOD 2)#0 THEN 5550
5545 PRINT "EVEN": RETURN
5550 PRINT "ODD": RETURN
5560 IF (SW MOD 2)=0 THEN PRINT
    "ODD"
5565 IF (SW MOD 2)=1 THEN PRINT
    "EVEN"
5570 RETURN
6000 REM ========================
6001 REM =      INTRODUCTION     =
6002 REM ========================
6005 TEXT : CALL -936
6010 VTAB 2
6015 PRINT "  THIS PROGRAM WILL SIMUL
    ATE THE "
6020 PRINT "FIFTEEN PUZZLE.  FIRST I
    WILL MIX UP"
6030 PRINT "THE NUMBERS IN THE TRAY.
     YOUR GOAL"
6035 PRINT "WILL BE TO SEE IF YOU CAN
     GET THEM BACK"
6040 PRINT "IN ORDER AGAIN."
6045 PRINT "  YOU MOVE BY PRESSING ON
    E OF THE KEYS"
6050 PRINT "'R', 'L', 'U', OR 'D'.  T
    HE LETTER YOU"
6055 PRINT "PRESS INDICATES THE DIREC
    TION THAT"
6065 PRINT "THE PIECE WILL MOVE INTO
    THE HOLE."
6070 PRINT "  FOR EXAMPLE, IF YOU WIS
    H TO MOVE THE"
6075 PRINT "PIECE ABOVE THE HOLE DOWN
     TO THE HOLE,"
6080 PRINT "THEN PRESS 'D' FOR DOWN."

6085 PRINT "IF YOU WISH TO QUIT, PRES
    S 'Q' FOR QUIT"
6090 PRINT "AND IF YOU WANT TO START
    OVER, PRESS"
6095 PRINT "THE 'ESC' KEY TO ESCAPE F
    ROM YOUR"
6100 PRINT "CURRENT SITUATION."
6110 PRINT : PRINT "NOW PRESS ANY KEY
     TO BEGIN..."
6115 GOSUB WAIT
6199 RETURN
7000 REM ========================
7001 REM =         MIX           =
7002 REM = SCRAMBLE THE PIECES.  =
7003 REM ========================
7005 FOR SLOT=1 TO 16:PUZZLE(SLOT)
    =-1: NEXT SLOT
7006 FOR NUM=0 TO 15
7010 SLOT= RND (16)+1
7015 IF PUZZLE(SLOT)>=0 THEN 7010

7020 PUZZLE(SLOT)=NUM
7025 IF NUM=0 THEN HOLE=SLOT
7030 NEXT NUM
7049 RETURN
7100 REM ========================
7101 REM =         INIT          =
7102 REM = DRAW INITIAL PICTURE =
7103 REM ========================
7105 GR :ROW=0:COL=0
7110 FOR PLACE=1 TO 16
7115 GOSUB CONVERT
7120 SNUM=PUZZLE(PLACE)
7130 GOSUB ERASE+50*SNUM
7135 NEXT PLACE
7140 REM ========================
7141 REM = INIT R,L,D,U ARRAYS =
7142 REM ========================
7145 FOR I=1 TO 16
7146 IF (I MOD 4)=1 THEN R(I)=-1

7147 IF (I MOD 4)#1 THEN R(I)=I-
    1
7148 NEXT I
7150 FOR I=1 TO 16
7151 IF (I MOD 4)=0 THEN L(I)=-1

7152 IF (I MOD 4)#0 THEN L(I)=I+
    1
```

LISTING 4.1 (*cont.*)

```
7153 NEXT I
7155 FOR I=1 TO 12
7156 U(I)=I+4
7157 NEXT I
7158 FOR I=13 TO 16:U(I)=-1: NEXT
     I
7160 FOR I=1 TO 4:D(I)=-1: NEXT
     I
7161 FOR I=5 TO 16
7162 D(I)=I-4
7163 NEXT I
7199 RETURN
7200 REM ==================
7201 REM =  WAIT ROUTINE  =
7202 REM ==================
7210 KEY= PEEK (KBD): IF KEY<128
      THEN 7210
7215 POKE CLR,0
7249 RETURN
7300 REM ========================
7301 REM =         CONVERT      =
7302 REM =                      =
7303 REM = CHANGE BOARD LOCATION =
7304 REM = NUMBERS INTO ROW AND  =
7305 REM = COL VALUES.           =
7306 REM ========================
7310 ROW=((PLACE-1)/4)*10
7315 COL=((PLACE-1) MOD 4)*10
7349 RETURN
8000 REM ============================
     =
8001 REM = ERASE SUBROUTINE FOLLOWED
     =
8002 REM = BY ROUTINES FOR THE
     =
8003 REM = FIFTEEN PUZZLE PIECES.
     =
8004 REM ============================
     =
8005 COLOR=SNUM
8008 COLOR=SNUM
8010 FOR LINE=0 TO 9: HLIN COL,COL+
     9 AT ROW+LINE: NEXT LINE
8015 COLOR=0
8049 RETURN
8050 REM ====================
8051 REM = DRAW A "1" IN THE =
8052 REM = HOLE POSITION.    =
8053 REM ====================
8055 GOSUB ERASE
8065 VLIN ROW+2,ROW+7 AT COL+5
8070 PLOT COL+4,ROW+3
8074 HLIN COL+3,COL+6 AT ROW+7
8099 RETURN
8100 REM ====================
8101 REM = DRAW A "2" IN THE =
8102 REM = HOLE POSITION.    =
8104 REM ====================
8105 GOSUB ERASE
8110 HLIN COL+3,COL+6 AT ROW+2
8115 HLIN COL+3,COL+6 AT ROW+5
8120 HLIN COL+3,COL+6 AT ROW+7
8125 VLIN ROW+2,ROW+5 AT COL+6
8130 PLOT COL+3,ROW+6
8135 PLOT COL+3,ROW+3
8149 RETURN
8150 REM ====================
8151 REM = DRAW A "3" IN THE =
8152 REM = HOLE POSITION.    =
8154 REM ====================
8155 GOSUB ERASE
8160 HLIN COL+3,COL+6 AT ROW+2
8165 HLIN COL+4,COL+6 AT ROW+4
8170 HLIN COL+3,COL+6 AT ROW+7
8175 VLIN ROW+2,ROW+7 AT COL+6
8180 PLOT COL+3,ROW+6
8199 RETURN
8200 REM ====================
8201 REM = DRAW A "4" IN THE =
8202 REM = HOLE POSITION.    =
8203 REM ====================
8205 GOSUB ERASE
8210 VLIN ROW+2,ROW+7 AT COL+5
8215 HLIN COL+3,COL+6 AT ROW+5
8220 PLOT COL+4,ROW+3
8225 PLOT COL+3,ROW+4
8249 RETURN
8250 REM ====================
8251 REM = DRAW A "5" IN THE =
8252 REM = HOLE POSITION.    =
8253 REM ====================
8255 GOSUB ERASE
8260 HLIN COL+3,COL+6 AT ROW+2
8265 HLIN COL+3,COL+6 AT ROW+4
8270 HLIN COL+3,COL+6 AT ROW+7
8275 PLOT COL+3,ROW+3
8280 VLIN ROW+4,ROW+7 AT COL+6
8285 PLOT COL+3,ROW+6
8299 RETURN
8300 REM ====================
8301 REM = DRAW A "6" IN THE =
8302 REM = HOLE POSITION.    =
8303 REM ====================
8305 GOSUB ERASE
8310 VLIN ROW+2,ROW+7 AT COL+3
8315 HLIN COL+3,COL+6 AT ROW+2
8320 HLIN COL+3,COL+6 AT ROW+5
8325 HLIN COL+3,COL+6 AT ROW+7
```

```
8330 PLOT COL+6,ROW+3
8335 PLOT COL+6,ROW+6
8349 RETURN
8350 REM ======================
8351 REM = DRAW A "7" IN THE =
8352 REM = HOLE POSITION.    =
8353 REM ======================
8355 GOSUB ERASE
8360 HLIN COL+3,COL+6 AT ROW+2
8365 PLOT COL+6,ROW+3
8370 PLOT COL+5,ROW+4
8375 PLOT COL+4,ROW+5
8380 VLIN ROW+6,ROW+7 AT COL+3
8399 RETURN
8400 REM ======================
8401 REM = DRAW AN "8" IN THE =
8402 REM = HOLE POSITION.    =
8403 REM ======================
8405 GOSUB ERASE
8410 HLIN COL+3,COL+6 AT ROW+2
8415 HLIN COL+3,COL+6 AT ROW+4
8420 HLIN COL+3,COL+6 AT ROW+5
8425 HLIN COL+3,COL+6 AT ROW+7
8430 VLIN ROW+2,ROW+7 AT COL+3
8435 VLIN ROW+2,ROW+7 AT COL+6
8449 RETURN
8450 REM ======================
8451 REM = DRAW A "9" IN THE =
8452 REM = HOLE POSITION.    =
8453 REM ======================
8455 GOSUB ERASE
8460 HLIN COL+4,COL+5 AT ROW+2
8465 HLIN COL+3,COL+6 AT ROW+5
8470 HLIN COL+3,COL+6 AT ROW+7
8475 VLIN ROW+3,ROW+7 AT COL+6
8480 VLIN ROW+3,ROW+4 AT COL+3
8499 RETURN
8500 REM ======================
8501 REM = DRAW A "10" IN THE =
8502 REM = HOLE POSITION.    =
8503 REM ======================
8505 GOSUB ERASE
8510 VLIN ROW+2,ROW+7 AT COL+2
8515 VLIN ROW+3,ROW+6 AT COL+5
8517 VLIN ROW+3,ROW+6 AT COL+8
8520 HLIN COL+1,COL+3 AT ROW+7
8525 HLIN COL+6,COL+7 AT ROW+2
8530 HLIN COL+6,COL+7 AT ROW+7
8535 PLOT COL+1,ROW+3
8549 RETURN
8550 REM ======================
8551 REM = DRAW AN "11" IN THE =
8552 REM = HOLE POSITION.    =
8553 REM ======================
8555 GOSUB ERASE
8560 VLIN ROW+2,ROW+7 AT COL+3
8565 VLIN ROW+2,ROW+7 AT COL+7
8570 HLIN COL+2,COL+4 AT ROW+7
8575 HLIN COL+6,COL+8 AT ROW+7
8580 PLOT COL+2,ROW+3
8585 PLOT COL+6,ROW+3
8599 RETURN
8600 REM ======================
8601 REM = DRAW A "12" IN THE =
8602 REM = HOLE POSITION.    =
8603 REM ======================
8605 GOSUB ERASE
8610 VLIN ROW+2,ROW+7 AT COL+3
8615 HLIN COL+2,COL+4 AT ROW+7
8620 PLOT COL+2,ROW+3
8625 HLIN COL+6,COL+8 AT ROW+2
8630 HLIN COL+6,COL+8 AT ROW+5
8635 HLIN COL+6,COL+8 AT ROW+7
8640 VLIN ROW+2,ROW+5 AT COL+8
8645 PLOT COL+6,ROW+6
8649 RETURN
8650 REM ======================
8651 REM = DRAW A "13" IN THE =
8652 REM = HOLE POSITION.    =
8653 REM ======================
8655 GOSUB ERASE
8660 VLIN ROW+2,ROW+7 AT COL+3
8665 HLIN COL+2,COL+4 AT ROW+7
8670 PLOT COL+2,ROW+3
8675 HLIN COL+6,COL+8 AT ROW+2
8677 HLIN COL+7,COL+8 AT ROW+4
8680 HLIN COL+6,COL+8 AT ROW+7
8685 VLIN ROW+2,ROW+7 AT COL+8
8690 PLOT COL+6,ROW+6
8699 RETURN
8700 REM ======================
8701 REM = DRAW A "14" IN THE =
8702 REM = HOLE POSITION.    =
8703 REM ======================
8705 GOSUB ERASE
8710 VLIN ROW+2,ROW+7 AT COL+3
8715 HLIN COL+2,COL+4 AT ROW+7
8720 PLOT COL+2,ROW+3
8725 VLIN ROW+2,ROW+7 AT COL+7
8730 HLIN COL+5,COL+8 AT ROW+5
8735 PLOT COL+6,ROW+3
8740 PLOT COL+5,ROW+4
8749 RETURN
8750 REM ======================
8751 REM = DRAW A "15" IN THE =
8752 REM = HOLE POSITION.    =
8753 REM ======================
8755 GOSUB ERASE
```

LISTING 4.1 *(cont.)*

```
8760 VLIN ROW+2,ROW+7 AT COL+3
8765 HLIN COL+2,COL+4 AT ROW+7
8770 PLOT COL+2,ROW+3
8775 HLIN COL+5,COL+8 AT ROW+2
8777 HLIN COL+5,COL+8 AT ROW+4
8779 HLIN COL+6,COL+8 AT ROW+7
8780 PLOT COL+5,ROW+3
8785 VLIN ROW+4,ROW+7 AT COL+8
8799 RETURN

]PR#0
]
```

LISTING 4.2 SEARCH FOR SOLUTION

```
]LIST
    1 REM ============================
    2 REM =                            =
    3 REM =  FIFTEEN PUZZLE SEARCH     =
    4 REM =                            =
    5 REM =DR. RICHARD C. VILE, JR.    =
    6 REM = ALL COMMERCIAL RIGHTS      =
    7 REM =        RESERVED            =
    8 REM =                            =
    9 REM ============================
   10 KBD=-16384:CLR=-16368
   15 DIM PIECE(16),A$(10),MOV$(50
      )
   16 DIM R(16),L(16),U(16),D(16)

   17 DIM WHROW(16),WHCOL(16)
   20 MIX=7000:INIT=7100:INTRO=6000

   21 MOVE=5000:ERASE=8000
   22 WAIT=7200:CONVERT=7300
   23 MOVR=5100:MOVL=5200:MOVUP=5300
      :MOVD=5400
   24 OK=5110:PARITY=5500:RESTART=
      5600
   25 WHERE=6200:MOVEIT=6300:SEARCH=
      2000
   26 UPROWS=3000:DOWNROWS=3100:LEFTCO
      LS=3200:RIGHTCOLS=3300
   27 INVERT=4000:FIRST=2200:SMOVE=
      2900
   50 GOSUB INTRO
   55 GOSUB MIX
   60 GOSUB INIT
   70 GOSUB SEARCH
   75 IF FLAG=1 THEN 90
   80 GOTO 70
   90 FLAG=0
   91 VTAB 24: POKE 50,63: PRINT
      "PRESS ANY KEY TO CONTINUE"
      ;: GOSUB WAIT
   92 POKE 50,255: GOSUB RESTART
   95 GOTO 55
 2000 REM ============================
      =
```

```
2001 REM =     SEARCH FOR SOLUTION
     =
2002 REM =
     =
2003 REM = ASSERT: WHROW AND WHCOL
     =
2004 REM =  HAVE BEEN INITIALIZED.
     =
2005 REM ============================
     =
2009 MOVER=0
2010 FOR I1=1 TO 16
2015 IF PIECE(I1)#I1 THEN 2025
2020 NEXT I1
2025 MOVER=I1: IF MOVER>=14 THEN
     RETURN
2050 PLACE=MOVER
2055 GOSUB CONVERT
2060 DROW=ROW/10:DCOL=COL/10: REM ***
     ** DESTINATION ROW AND COLUMN **
     ***
2065 SROW=WHROW(I1):SCOL=WHCOL(I1)
     : REM ***** WHERE IT IS NOW ****
     *
2070 PLACE=HOLE: GOSUB CONVERT
2071 HROW=ROW/10:HCOL=COL/10
2075 GOTO (DCOL>SCOL)*2085+(DCOL<
     SCOL)*2080+(DCOL=SCOL)*2090

2078 GOTO 2100
2080 GOTO (HCOL=SCOL)*3000+(HCOL<
     SCOL)*3300+(HCOL>SCOL)*3600

2085 GOSUB FIRST: GOTO 2100
2090 IF HCOL>=SCOL THEN 2080
2091 IF HROW=SROW THEN 2095
2092 KEY= ASC("L"): GOSUB MOVE: GOTO
     2100
2095 IF HROW=4 THEN 2098
2096 MOV$="ULLD": GOSUB SMOVE: GOTO
     2100
2098 MOV$="DLLU": GOSUB SMOVE
2100 GOSUB INVERT
2105 GOTO 2005
2200 REM ============================
2201 REM =           FIRST           =
2202 REM =                           =
2203 REM = DESTINATION COLUMN IS    =
2204 REM = GREATER THAN SOURCE.     =
2205 REM ============================
2210 GOTO (HROW=SROW)*2300+(HROW<
     SROW)*2400+(HROW>SROW)*2500
```

LISTING 4.2 *(cont.)*

```
2300 REM
2310 IF SCOL>HCOL THEN 2320
2315 KEY= ASC("R"): GOSUB MOVE: RETURN
2320 IF ABS (HCOL-SCOL)=1 THEN 2330

2325 KEY= ASC("L"): GOSUB MOVE: RETURN
2330 IF HROW=3 THEN 2350
2335 MOV$="ULLD"
2340 GOSUB SMOVE: RETURN
2350 MOV$="DLLU": GOSUB SMOVE: RETURN

2400 REM
2410 IF HCOL=SCOL THEN 2430
2420 KEY= ASC("U"): GOSUB MOVE: RETURN
2430 IF ABS (HROW-SROW)=1 THEN 2450

2440 KEY= ASC("U"): GOSUB MOVE: RETURN
2450 MOV$="LUR": GOSUB SMOVE: RETURN

2500 REM
2510 IF HCOL=SCOL THEN 2530
2520 KEY= ASC("D"): GOSUB MOVE: RETURN
2530 IF ABS (HROW-SROW)=1 THEN 2550

2540 KEY= ASC("D"): GOSUB MOVE: RETURN
2550 MOV$="LDR": GOSUB SMOVE: RETURN

2900 REM   STRING MOVE INTERPRETER
2901 REM
2910 FOR I2=1 TO LEN(MOV$)
2915 CH$=MOV$(I2,I2)
2920 KEY= ASC(CH$)
2930 GOSUB MOVE
2935 NEXT I2
2999 RETURN
3000 REM ==============================

3001 REM =           UPROWS           =

3002 REM =                            =

3003 REM = HOLE IN SAME COLUMN AS     =

3004 REM = MOVER.                     =

3005 REM ==============================

3008 IF HROW<SROW THEN 3100
3010 IF HCOL>0 AND DCOL#SCOL THEN
     KEY= ASC("R")
3012 IF HCOL>0 AND DCOL=SCOL THEN
     KEY= ASC("L")
3015 IF HCOL=0 THEN KEY= ASC("L"
     )
```

```
3020 GOSUB MOVE: GOTO 2100
3100 KEY= ASC("U"): GOSUB MOVE: GOTO
     2100
3300 REM   HOLE COLUMN TO LEFT OF MOV
     ER COLUMN
3305 IF HROW>SROW THEN KEY= ASC(
     "D")
3310 IF HROW<=SROW THEN KEY= ASC(
     "L")
3315 REM   IF HROW<SROW THEN KEY=ASC(
     "U")
3320 GOSUB MOVE: GOTO 2100
3600 REM   HOLE COLUMN TO RIGHT OF MO
     VER COLUMN
3605 GOTO (HROW=SROW)*3700+(HROW<
     SROW)*3800+(HROW>SROW)*3610

3610 IF SROW=DROW THEN KEY= ASC(
     "R")
3612 IF SROW<DROW THEN KEY= ASC(
     "D")
3615 IF SROW>DROW THEN KEY= ASC(
     "D"): GOSUB MOVE
3620 GOTO 2100
3700 IF HROW>0 THEN KEY= ASC("D"
     )
3705 IF HROW=0 THEN KEY= ASC("U"
     )
3710 GOSUB MOVE: GOTO 2100
3800 KEY= ASC("R"): GOSUB MOVE: GOTO
     2100
4000 REM ============================
4001 REM =          INVERT          =
4002 REM =                          =
4003 REM = FORM INVERTED LIST OF    =
4004 REM = ROW AND COLUMNS NUMBERS=
4005 REM = IN ARRAYS WHROW AND      =
4006 REM = WHCOL.                   =
4007 REM ============================
4010 FOR I1=1 TO 16
4015 PLACE=I1
4020 GOSUB CONVERT
4025 WHO=PIECE(I1)
4030 WHROW(WHO)=ROW/10
4035 WHCOL(WHO)=COL/10
4040 NEXT I1
4099 RETURN
5000 REM ============================
5001 REM =           MOVE           =
5002 REM ============================
5004 DONE=0
5005 IF DONE THEN RETURN
5010 IF KEY# ASC("R") THEN 5020
5015 GOSUB MOVR: GOTO 5005
5020 IF KEY# ASC("L") THEN 5030
5025 GOSUB MOVL: GOTO 5005
5030 IF KEY# ASC("U") THEN 5040
5035 GOSUB MOVUP: GOTO 5005
5040 IF KEY# ASC("D") THEN 5050
5045 GOSUB MOVD: GOTO 5005
5050 IF KEY# ASC("Q") THEN 5060
5055 TEXT : CALL -936: END
5060 IF KEY#155 THEN 5070
5065 TEXT : CALL -936:FLAG=1: RETURN

5070 IF KEY# ASC("H") THEN 5080
5075 GOSUB PARITY: GOTO 5005
5080 CALL -198: REM  BEEP
5085 GOTO 5005
5100 REM ========================
5101 REM =         MOVR         =
5102 REM = TRY TO MOVE RIGHT    =
5103 REM = INTO THE HOLE.       =
5104 REM ========================
5105 SLOC=R(HOLE): IF SLOC#-1 THEN
     5110
5108 CALL -198: RETURN
5110 REM ========================
5111 REM =          OK           =
5112 REM = MOVE IS LEGAL - IT'S =
5113 REM = OK TO MAKE IT.       =
5114 REM ========================
5115 PLACE=HOLE
5120 GOSUB CONVERT
5125 SNUM=PIECE(SLOC)
5130 GOSUB ERASE+SNUM*50
5140 PLACE=SLOC
5145 GOSUB CONVERT
5160 REM  UPDATE HOLE INFO
5165 PIECE(HOLE)=SNUM
5170 PIECE(SLOC)=0
5175 HOLE=SLOC
5185 SNUM=0: GOSUB ERASE
5190 DONE=1
5199 RETURN
5200 REM ========================
5201 REM =         MOVL          =
5202 REM = TRY TO MOVE LEFT     =
5203 REM = INTO THE HOLE.       =
5204 REM ========================
5205 SLOC=L(HOLE): IF SLOC#-1 THEN
     5210
5207 CALL -198: RETURN
5210 GOSUB OK
5299 RETURN
5300 REM ========================
5301 REM =         MOVUP         =
5302 REM = TRY TO MOVE UP INTO =
```

LISTING 4.2 *(cont.)*

```
5303 REM = THE HOLE.              =
5304 REM ========================
5305 SLOC=U(HOLE): IF SLOC#-1 THEN
     5310
5307 CALL -198: RETURN
5310 GOSUB OK
5399 RETURN
5400 REM ========================
5401 REM =        MOVD           =
5402 REM = TRY TO MOVE DOWN      =
5403 REM = INTO THE HOLE.        =
5404 REM ========================
5405 SLOC=D(HOLE): IF SLOC#-1 THEN
     5410
5407 CALL -198: RETURN
5410 GOSUB OK
5499 RETURN
5500 REM  CALCULATE PARITY OF THE
5501 REM   PERMUTATION
5502 PRINT : PRINT "     HELP IS ON T
     HE WAY"
5505 SW=0
5510 FOR X=1 TO 15
5515 FOR Y=X+1 TO 16
5520 XX=PIECE(X):YY=PIECE(Y)
5525 IF XX=0 THEN XX=16: IF YY=0
      THEN YY=16
5530 IF XX>YY THEN SW=SW+1
5532 VTAB 24: TAB 10: PRINT "***"
     ;: VTAB 24: TAB 10: PRINT "   "
     ;
5535 NEXT Y,X
5537 IF HOLE=2 OR HOLE=4 OR HOLE=
     5 OR HOLE=7 OR HOLE=10 OR HOLE=
     12 OR HOLE=13 OR HOLE=15 THEN
     5560
5540 IF (SW MOD 2)#0 THEN 5550
5545 PRINT "EVEN": RETURN
5550 PRINT "ODD": RETURN
5560 IF (SW MOD 2)=0 THEN PRINT
     "ODD"
5565 IF (SW MOD 2)=1 THEN PRINT
     "EVEN"
5570 RETURN
6000 REM ========================
6001 REM =     INTRODUCTION      =
6002 REM ========================
6005 TEXT : CALL -936
6010 VTAB 2
6015 PRINT "  THIS PROGRAM WILL SIMUL
     ATE THE "
6020 PRINT "FIFTEEN PUZZLE.  FIRST I
     WILL MIX UP"
6030 PRINT "THE NUMBERS IN THE TRAY.
     THEN I WILL"
6035 PRINT "ATTEMPT TO MOVE THEM BACK
      INTO THE"
6040 PRINT "CORRECT ORDER AGAIN."

6110 PRINT : PRINT "NOW PRESS ANY KEY
      TO BEGIN..."
6115 GOSUB WAIT
6199 RETURN
6200 REM ============================

6201 REM =          DIFFER           =

6202 REM =  CALCULATE DIFFERENCE OF =

6203 REM = SOURCE AND DESTINATION   =

6204 REM = ROW OR COLUMN - MULTIPLE =

6205 REM = MOVE ALONG SAME ROW OR   =

6206 REM = COLUMN CONTEMPLATED.     =

6207 REM ============================

6210 PLACE=HOLE
6215 GOSUB CONVERT
6220 ROWDIFF=X-ROW/10
6225 COLDIFF=Y-COL/10
6240 IF COLDIFF>0 THEN KEY= ASC(
     "L")
6245 I2= ABS (COLDIFF): GOSUB MOVEIT
6250 IF ROWDIFF>0 THEN KEY= ASC(
     "U")
6255 IF ROWDIFF<0 THEN KEY= ASC(
     "D")
6260 I2= ABS (ROWDIFF): GOSUB MOVEIT
6299 RETURN
6300 REM ========================
6301 REM =          MOVEIT        =
6302 REM =                        =
6303 REM = MOVE IN SAME DIRECTION =
6304 REM = MULTIPLE TIMES.        =
6305 REM ========================
6320 IF I2<=0 THEN RETURN
6325 FOR I1=1 TO I2
6330 GOSUB MOVE
6335 NEXT I1
6399 RETURN
7000 REM ========================
7001 REM =         MIX            =
7002 REM = SCRAMBLE THE PIECES.   =
7003 REM ========================
7005 FOR SLOT=1 TO 16:PIECE(SLOT)
     =-1: NEXT SLOT
7006 FOR NUM=0 TO 15
```

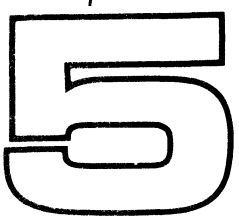
```
7010 SLOT= RND (16)+1
7015 IF PIECE(SLOT)>=0 THEN 7010

7020 PIECE(SLOT)=NUM
7025 IF NUM=0 THEN HOLE=SLOT
7030 NEXT NUM
7049 RETURN
7100 REM ========================
7101 REM =         INIT         =
7102 REM = DRAW INITIAL PICTURE =
7103 REM ========================
7105 GR :ROW=0:COL=0
7110 FOR PLACE=1 TO 16
7115 GOSUB CONVERT
7120 SNUM=PIECE(PLACE)
7130 GOSUB ERASE+50*SNUM
7135 NEXT PLACE
7140 REM ========================
7141 REM = INIT R,L,D,U ARRAYS =
7142 REM ========================
7145 FOR I=1 TO 16
7146 IF (I MOD 4)=1 THEN R(I)=-1

7147 IF (I MOD 4)#1 THEN R(I)=I-
     1
7148 NEXT I
7150 FOR I=1 TO 16
7151 IF (I MOD 4)=0 THEN L(I)=-1

7152 IF (I MOD 4)#0 THEN L(I)=I+
     1
7153 NEXT I
7155 FOR I=1 TO 12
7156 U(I)=I+4
7157 NEXT I
7158 FOR I=13 TO 16:U(I)=-1: NEXT
     I
7160 FOR I=1 TO 4:D(I)=-1: NEXT
     I
7161 FOR I=5 TO 16
7162 D(I)=I-4
7163 NEXT I
7170 GOSUB INVERT
7199 RETURN
7200 REM ==================
7201 REM =  WAIT ROUTINE  =
7202 REM ==================
7210 KEY= PEEK (KBD): IF KEY<128
      THEN 7210
7215 POKE CLR,0
7249 RETURN
7300 REM ========================
7301 REM =         CONVERT      =
7302 REM =                      =
7303 REM = CHANGE BOARD LOCATION =
7304 REM = NUMBERS INTO ROW AND  =
7305 REM = COL VALUES.           =
7306 REM ========================
7310 ROW=((PLACE-1)/4)*10
7315 COL=((PLACE-1) MOD 4)*10
7349 RETURN
8000 REM ============================
     =
8001 REM = ERASE SUBROUTINE FOLLOWED
     =
8002 REM = BY ROUTINES FOR THE
     =
8003 REM = FIFTEEN PUZZLE PIECES.
     =
8004 REM ============================
     =
8005 COLOR=SNUM
8008 COLOR=SNUM
8010 FOR LINE=0 TO 9: HLIN COL,COL+
     9 AT ROW+LINE: NEXT LINE
8015 COLOR=0
8049 RETURN
8050 REM ====================
8051 REM = DRAW A "1" IN THE =
8052 REM = HOLE POSITION.    =
8053 REM ====================
8055 GOSUB ERASE
8065 VLIN ROW+2,ROW+7 AT COL+5
8070 PLOT COL+4,ROW+3
8074 HLIN COL+3,COL+6 AT ROW+7
8099 RETURN
8100 REM ====================
8101 REM = DRAW A "2" IN THE =
8102 REM = HOLE POSITION.    =
8104 REM ====================
8105 GOSUB ERASE
8110 HLIN COL+3,COL+6 AT ROW+2
8115 HLIN COL+3,COL+6 AT ROW+5
8120 HLIN COL+3,COL+6 AT ROW+7
8125 VLIN ROW+2,ROW+5 AT COL+6
8130 PLOT COL+3,ROW+6
8135 PLOT COL+3,ROW+3
8149 RETURN
8150 REM ====================
8151 REM = DRAW A "3" IN THE =
8152 REM = HOLE POSITION.    =
8154 REM ====================
8155 GOSUB ERASE
8160 HLIN COL+3,COL+6 AT ROW+2
8165 HLIN COL+4,COL+6 AT ROW+4
8170 HLIN COL+3,COL+6 AT ROW+7
8175 VLIN ROW+2,ROW+7 AT COL+6
8180 PLOT COL+3,ROW+6
8199 RETURN
8200 REM ====================
```

LISTING 4.2 (*cont.*)

```
8201 REM = DRAW A "4" IN THE =
8202 REM = HOLE POSITION.    =
8203 REM ====================
8205 GOSUB ERASE
8210 VLIN ROW+2,ROW+7 AT COL+5
8215 HLIN COL+3,COL+6 AT ROW+5
8220 PLOT COL+4,ROW+3
8225 PLOT COL+3,ROW+4
8249 RETURN
8250 REM ====================
8251 REM = DRAW A "5" IN THE =
8252 REM = HOLE POSITION.    =
8253 REM ====================
8255 GOSUB ERASE
8260 HLIN COL+3,COL+6 AT ROW+2
8265 HLIN COL+3,COL+6 AT ROW+4
8270 HLIN COL+3,COL+6 AT ROW+7
8275 PLOT COL+3,ROW+3
8280 VLIN ROW+4,ROW+7 AT COL+6
8285 PLOT COL+3,ROW+6
8299 RETURN
8300 REM ====================
8301 REM = DRAW A "6" IN THE =
8302 REM = HOLE POSITION.    =
8303 REM ====================
8305 GOSUB ERASE
8310 VLIN ROW+2,ROW+7 AT COL+3
8315 HLIN COL+3,COL+6 AT ROW+2
8320 HLIN COL+3,COL+6 AT ROW+5
8325 HLIN COL+3,COL+6 AT ROW+7
8330 PLOT COL+6,ROW+3
8335 PLOT COL+6,ROW+6
8349 RETURN
8350 REM ====================
8351 REM = DRAW A "7" IN THE =
8352 REM = HOLE POSITION.    =
8353 REM ====================
8355 GOSUB ERASE
8360 HLIN COL+3,COL+6 AT ROW+2
8365 PLOT COL+6,ROW+3
8370 PLOT COL+5,ROW+4
8375 PLOT COL+4,ROW+5
8380 VLIN ROW+6,ROW+7 AT COL+3
8399 RETURN
8400 REM =====================
8401 REM = DRAW AN "8" IN THE =
8402 REM = HOLE POSITION.    =
8403 REM =====================
8405 GOSUB ERASE
8410 HLIN COL+3,COL+6 AT ROW+2
8415 HLIN COL+3,COL+6 AT ROW+4
8420 HLIN COL+3,COL+6 AT ROW+5
8425 HLIN COL+3,COL+6 AT ROW+7
8430 VLIN ROW+2,ROW+7 AT COL+3
8435 VLIN ROW+2,ROW+7 AT COL+6
8449 RETURN
8450 REM ====================
8451 REM = DRAW A "9" IN THE =
8452 REM = HOLE POSITION.    =
8453 REM ====================
8455 GOSUB ERASE
8460 HLIN COL+4,COL+5 AT ROW+2
8465 HLIN COL+3,COL+6 AT ROW+5
8470 HLIN COL+3,COL+6 AT ROW+7
8475 VLIN ROW+3,ROW+7 AT COL+6
8480 VLIN ROW+3,ROW+4 AT COL+3
8499 RETURN
8500 REM =====================
8501 REM = DRAW A "10" IN THE =
8502 REM = HOLE POSITION.     =
8503 REM =====================
8505 GOSUB ERASE
8510 VLIN ROW+2,ROW+7 AT COL+2
8515 VLIN ROW+3,ROW+6 AT COL+5
8517 VLIN ROW+3,ROW+6 AT COL+8
8520 HLIN COL+1,COL+3 AT ROW+7
8525 HLIN COL+6,COL+7 AT ROW+2
8530 HLIN COL+6,COL+7 AT ROW+7
8535 PLOT COL+1,ROW+3
8549 RETURN
8550 REM =====================
8551 REM = DRAW AN "11" IN THE =
8552 REM = HOLE POSITION.     =
8553 REM =====================
8555 GOSUB ERASE
8560 VLIN ROW+2,ROW+7 AT COL+3
8565 VLIN ROW+2,ROW+7 AT COL+7
8570 HLIN COL+2,COL+4 AT ROW+7
8575 HLIN COL+6,COL+8 AT ROW+7
8580 PLOT COL+2,ROW+3
8585 PLOT COL+6,ROW+3
8599 RETURN
8600 REM =====================
8601 REM = DRAW A "12" IN THE =
8602 REM = HOLE POSITION.     =
8603 REM =====================
8605 GOSUB ERASE
8610 VLIN ROW+2,ROW+7 AT COL+3
8615 HLIN COL+2,COL+4 AT ROW+7
8620 PLOT COL+2,ROW+3
8625 HLIN COL+6,COL+8 AT ROW+2
8630 HLIN COL+6,COL+8 AT ROW+5
8635 HLIN COL+6,COL+8 AT ROW+7
8640 VLIN ROW+2,ROW+5 AT COL+8
8645 PLOT COL+6,ROW+6
8649 RETURN
8650 REM =====================
8651 REM = DRAW A "13" IN THE =
```

```
8652 REM = HOLE POSITION.      =
8653 REM ======================
8655 GOSUB ERASE
8660 VLIN ROW+2,ROW+7 AT COL+3
8665 HLIN COL+2,COL+4 AT ROW+7
8670 PLOT COL+2,ROW+3
8675 HLIN COL+6,COL+8 AT ROW+2
8677 HLIN COL+7,COL+8 AT ROW+4
8680 HLIN COL+6,COL+8 AT ROW+7
8685 VLIN ROW+2,ROW+7 AT COL+8
8690 PLOT COL+6,ROW+6
8699 RETURN
8700 REM ======================
8701 REM = DRAW A "14" IN THE =
8702 REM = HOLE POSITION.      =
8703 REM ======================
8705 GOSUB ERASE
8710 VLIN ROW+2,ROW+7 AT COL+3
8715 HLIN COL+2,COL+4 AT ROW+7
8720 PLOT COL+2,ROW+3
8725 VLIN ROW+2,ROW+7 AT COL+7
8730 HLIN COL+5,COL+8 AT ROW+5
8735 PLOT COL+6,ROW+3
8740 PLOT COL+5,ROW+4
8749 RETURN
8750 REM ======================
8751 REM = DRAW A "15" IN THE =
8752 REM = HOLE POSITION.      =
8753 REM ======================
8755 GOSUB ERASE
8760 VLIN ROW+2,ROW+7 AT COL+3
8765 HLIN COL+2,COL+4 AT ROW+7
8770 PLOT COL+2,ROW+3
8775 HLIN COL+5,COL+8 AT ROW+2
8777 HLIN COL+5,COL+8 AT ROW+4
8779 HLIN COL+6,COL+8 AT ROW+7
8780 PLOT COL+5,ROW+3
8785 VLIN ROW+4,ROW+7 AT COL+8
8799 RETURN
```

Chapter



APPLE DOS: Using Files in Programs

1. REVIEW OF DOS USAGE

The APPLE II Disk Operating System or DOS allows BASIC programmers to make use of disk files. These store information such as programs or data. DOS files are classified into four categories:

I Integer BASIC programs
A APPLESOFT BASIC programs
B Binary information; machine-language programs or pure binary data
T Text files: ASCII text

The first three of these filetypes are used to store programs and their contents may not be accessed for manipulation by a running BASIC program. On the other hand, the T-type files contain *text* or pure information. BASIC programs may create such files, and open, close, read, and write them.

The Role of Control-D

Since the APPLE II was released before DOS was designed, specific DOS commands were not incorporated into Integer BASIC. The DOS provides a clever way around this limitation, however. The data which is written by a BASIC PRINT statement is intercepted by DOS and examined. When DOS does this and finds a Control-D character in the stream of information, it picks up the rest of the output line and interprets it as a command to itself. For example, the short BASIC segment:

```
10 D$="":REM CONTROL-D
20 PRINT D$;"CATALOG"
30 END
```

will cause DOS to list the Disk Catalog to the screen when it is run. This is what DOS "sees" in the output stream:

```
(↑D) CATALOG
```

(where (↑D) represents the Control-D character). (↑D) is DOS's signal to execute the remainder of the output, in this case "CATALOG," as a DOS command. Using this approach, many capabilities for manipulating files have been added to Integer BASIC.

Creating a Text File

There is no way to create a text file directly from the APPLE II keyboard. A program must be written which creates it for you. The magic command which does this is

OPEN. The reason that it is "magic" is that if you tell DOS to OPEN a file which isn't there, DOS makes it appear right before your very eyes!

```
10 D$="":REM CONTROL-D
20 PRINT D$;"OPEN NEW STUFF"
30 END
```

Running this program will make the file "NEW STUFF," of type T, appear in the disk catalog. It will be one sector long, empty, and quite uninteresting; yet it will be there.

Writing to and Reading from a Text File

Again, since Integer BASIC was written before DOS existed, there are no disk file input/output statements built into the language. However, the APPLE II Monitor ROM was carefully designed with the idea of expansion in mind. It is possible to switch I/O horses in the middle of the stream. All I/O goes through two routines in the Monitor: COUT and RDCHAR. Both of these routines provide a way to *hook up* to another routine when necessary.

DOS can hook its own input and output routines, for disk files, into the COUT and RDCHAR routines. The way it knows to do this is when it sees a READ or a WRITE command in the normal output stream, prefixed by Control-D. Once this occurs, all INPUT or PRINT statements in the corresponding BASIC program will use the file which was referenced. The statement:

```
PRINT D$;"READ RECIPES"
```

will cause all subsequent INPUT statements to get their data from the disk file RECIPES, rather than from the keyboard. The statement:

```
PRINT D$;"WRITE PAYROLL"
```

will cause all subsequent PRINT statements to go to the disk file PAYROLL as well as the terminal screen.

Of course, before either READ or WRITE can work, the files they reference must be OPENed.

Sample Programs: Text Creator and Text Lister

Two simple programs which illustrate these points are shown in Listings 5.1 and 5.2 at the end of the chapter. The first allows interactive creation of a TEXT file and the second simply lists such a file.

Symbolic end of file

When DOS tries to obtain more input from a file, but finds that the end of the file has been reached, it prints the error message:

```
END OF DATA
```

and causes the BASIC program which requested the input to terminate. Integer BASIC provides no way built into the language to trap this error. (APPLESOFT BASIC does have this capability as we shall see later on.) It is possible to do it, but it involves technical knowledge of the innards of DOS and BASIC and we shall not go into that here.

There is a way for the programmer to handle the END OF DATA problem, however. The use of a symbolic end of file is recommended. A symbolic end of file is a piece of data which is not to be used in any calculations made by a program reading the file, but rather serves only to tell the program not to expect more data. The program can test for this *special* piece of data and avoid the embarrassment of getting the END OF DATA slap in the face. Listing 5.3 at the end of the chapter shows a modification of the Text Lister program which uses this idea. It looks for the string EOF and when it finds it, it stops reading from the file.

There is a slight problem here, in that someday you might want to store the string EOF in a file for reasons other than using it as a symbolic end of file. Should this remote possibility arise, you may simply modify the test to look for some other string as symbolic EOF.

Cancelling DOS Commands

The use of the Control-D commands is very simple minded and paradoxically for that reason can get very confusing if you make the wrong assumptions. There is one very important general rule to keep in mind:

> Any DOS command cancels the previous DOS command.

Frame this and hang it over your bed so you will see it as you say your prayers every night.

What this means is that it may not be adequate to issue one command such as:

```
PRINT D$;"READ BOZO"
```

in a program, and blithely assume that *all* subsequent INPUT statements will forever reference the file BOZO. This will only be true as long as you don't issue another DOS command. The minute you do something like

```
PRINT D$;"WRITE THE CLOWN"
```

your hookup to BOZO will be lost, and it will be necessary to issue the

```
PRINT D$;"READ BOZO"
```

all over again.

There is one exception to this rule: DOS commands for READing and WRITEing do not cause an OPEN file to be CLOSEd. Obviously if this were the case, we would never be able to READ or WRITE files in the first place. The general rule really should be formulated as "Any DOS command will cause the input/output hooks to revert to BASIC." Since this is a bit harder to understand at first, we used the simpler version shown previously for bedtime consumption.

Use strings to store filenames

Using statements like:

```
PRINT D$;"READ BOZO"
```

severely limit the flexibility of programs. It is a good idea to *assume* that you will want to use a program with files that have different *names*. This being the case, you will want to make it as easy as possible to accommodate that goal. One way of making life easier for yourself is to use string variables to set up the names of files and then use the string variables in all DOS commands in the program:

```
FILE$ = "BOZO"
  . . .
PRINT D$;"OPEN ";FILE$
  . . .
PRINT D$;"READ ";FILE$
  . . .
PRINT D$;"CLOSE";FILE$
```

Then there is just one statement you need to change in order to switch to another file, instead of three or more.

Another good idea is to have the program *prompt* for the filename:

```
PRINT "WHAT IS THE NAME OF THE INPUT FILE?"
INPUT FILE$
```

Then the program may use different files on different runs *with no change* whatsoever to the program itself.

2. EXEC FILES

The APPLE DOS has the capability of reading commands to itself from TEXT files as well as from the keyboard. For example, if the TEXT file SETUP contained the following lines:

```
BLOAD BIG LETTERS
BLOAD APPLE-BITS
LOMEM: 4096
RUN BIG LETTERS DRIVER
```

then typing:

```
EXEC SETUP
```

would cause DOS to read all those commands from the file and execute them one at a time. This capability is obviously quite handy in cases like the one in the example above, where it is necessary to set up an environment for a program. Then the several required commands need not be typed each time you wish to run the program.

Using EXECs to create turnkey systems

The use of EXEC files can extend the capabilities of your HELLO programs. Have you ever wanted to use the HELLO program to run a program that had special "environment" requirements such as setting HIMEM or LOMEM, only to find that you couldn't get those commands to be accepted by a BASIC program?

You can get around this by storing all the commands you would normally issue manually in an EXEC file, such as the SETUP file described above, and then having the HELLO program issue the EXEC command:

```
10 PRINT D$;"EXEC SETUP"
20 END
```

The RUN command in the EXECed file will be stacked until the HELLO program terminates, and then automatically invoked.

Storing BASIC programs in TEXT files

The APPLE DOS manual introduces the idea of *capturing* BASIC programs using TEXT files. The following program may be used to accomplish this:

```
32750 DIM FILE$(30)
32751 D$="":REM CONTROL-D
32752 INPUT "CAPTURE FILE===>",FILE$
32754 PRINT D$;"OPEN ";FILE$
32756 PRINT D$;"WRITE ";FILE$
32758 POKE 33,30
32760 LIST 0,32749
32762 PRINT D$;"CLOSE ";FILE$
32764 TEXT : END
```

The values of the line numbers used in line 32760 will need changing, depending on the portion of the program you wish to capture. The capture program itself is invoked by:

RUN 32750

Presumably, it is tacked onto the end of the program being captured.

You are probably wondering what you can do with the resulting TEXT file once you have created it with the capture program. The answer is that you can EXEC it. When DOS reads a line that starts with a line number from an EXEC file, it is smart enough to realize that the line must contain a BASIC statement, and it passes the line on to the BASIC interpreter. So if you EXEC a captured program, the program winds up being fed back into memory, statement by statement.

This technique has a number of applications, some straightforward, and others a bit more bizarre.

- Loading a program from more than one file.

The LOAD or RUN commands cause a single file containing a BASIC program to be read into memory. Issuing another LOAD causes any program already in memory to be *blitzed*. It would seem impossible to store a program in two separate files for this reason. Yet with the capture program, this can be accomplished. If the capture program is run more than once and different ranges of lines are specified in line 32760, then we wind up with two or more TEXT files, each of which contains a separate "piece" of the program. To load the program back into memory, several EXEC statements may be used. To load only part of the program, fewer EXEC statements may be used. This could be useful in creating variations on a program, in which the common parts were stored in one EXEC file and the differing parts stored each in its own separate EXEC file.

- Incorporating standard subroutines into many different BASIC programs.

Earlier, we recommended the use of standard subroutines and skeleton programs. The technique of capture facilitates this quite nicely. First of all, you capture each new general-purpose subroutine which you are likely to reuse. You may even maintain a separate utility diskette for just those files. Second, you *reserve* a certain chunk of the line number space to accommodate those subroutines, in line with the skeleton program philosophy. You write those subroutines to occupy multiples of 50–100 lines each, and make sure that you don't reuse the same range of line numbers twice. When you begin work on a new application, you simply take out the utility diskette and EXEC in all the subroutines you need for the new program.

The program in Listing 5.4 automates this process. It puts up a menu listing the available subroutines and EXECs in the ones you need as you choose them. Note that it was coded to avoid the reserved area of line numbers, except for the standard routines that it itself uses. When finished using the loader program, you simply DEL the line numbers which it occupies and continue developing your new program. The TEXT files EXECed in by this program may be designed to "restart" the program—we comment on how this works below, after further discussion of CAPTURE.

- Making APPLESOFT programs from Integer BASIC programs.

This point is of interest because it was used by the author to provide a starting point for the APPLESOFT version of the APPLE Trivia Quiz. The way it worked was this:

1. The Integer BASIC version of the APPLE Trivia Quiz was captured in a TEXT file.
2. The FP command was issued to switch interpreters.
3. The captured program was EXECed into memory under APPLESOFT.

The only reason this works is that APPLESOFT does virtually no syntax checking as program statements are entered. This means that any program may be EXECed in as an APPLESOFT program. Going the other direction will not work as well—any APPLESOFT statements which are not legal Integer statements will cause a:

*****SYNTAX ERROR**

during the EXEC process and will not be part of the EXECed program.

Capture Capture!

No, this is not a rerun of Mary Hartmann, Mary Hartmann! The capture process requires that you key in the capture program at the tail end of every program you wish to capture. This can get annoying, especially with all those five digit line numbers! The solution is to key in CAPTURE once, and *run it on itself*. That is, use the line

32760 LIST 32750,32767

Let's suppose you do this and call the resulting TEXT file CAPTURE. Then whenever you want to capture some other program or subroutine, just say:

EXEC CAPTURE

to get the capture program back into memory. Don't forget to modify line 32760 to contain the desired range of line numbers. Finally,

RUN 32750

and you've bagged your game!

Restarting a program from an EXEC

When a statement of the form:

```
PRINT D$;"EXEC ";F$
```

is issued from a running program, it is not invoked until that program stops. Thus, it would seem that our subroutine loader program will only be able to load a single program at a time. This can be gotten around by making sure that the line:

```
RUN 14
```

occurs as the last line of each of the "subroutine EXECs." This will cause the LOADER program to be restarted after each subroutine is EXECed into memory.

There is one tricky point here. The FOR loop in line 10:

```
FOR I=1 TO 4:POKE I,0:NEXT I
```

is only executed when the program is first run. This is because these memory locations are used to keep track of which subroutines in the menu have already been loaded. The numbers of these subroutines are printed in inverse video when the program restarts each time. This explains why the RUN statement in the subroutine EXEC is RUN 14, instead of just plain RUN.

To create such subroutine EXECs, simply modify the CAPTURE program by adding line 32761:

```
32761 PRINT "RUN 14"
```

and capture it into a separate text file, say SUB CAPTURE, or some other suggestive title.

3. DUELLING DO LOOPS

The program in Listing 5.5 presents an educational application of EXEC files. The program presents an interactive game designed to assist in learning about BASIC FOR loops. (The name DO loops was chosen for obvious alliterative reasons!) The program prints a pattern of asterisks on the screen and the player must write a BASIC subroutine to duplicate it. The player is supposed to use only FOR statements and simple PRINT statements, such as:

```
PRINT
PRINT "*"
PRINT "*";
PRINT " "
```

The patterns are printed by 10 subroutines. The program selects among these at random and chooses the size parameters associated with the patterns at random as well.

The player is invited to key in the subroutine and the program accepts it and stores it in an EXEC file. After the subroutine is complete, the program issues a command to EXEC it into memory and then ENDs. Finally, the EXEC command is carried out by BASIC. This results in adding the player's subroutine to the program in memory. The subroutine has been generated so as not to interfere with the program already there. The last statement of the EXEC causes the program to be RUN at a statement number which invokes the subroutine.

There are some obvious limitations to the game, which most players will be willing to put up with:

- There is no syntax checking of the subroutine as it is entered, since it is being READ by a running BASIC program, not by a BASIC interpreter.
- There is no scoring. It would be quite difficult and involved to judge the results of the player's subroutine (though not impossible!).
- By the time the player's output appears on the screen, the original output is long gone. This requires the player to remember what the original pattern looked like! A possible way around this would be to output the patterns to a printer, as well as the screen (see the Explorations section).

4. EXPLORATIONS

- Modify the Duelling DO Loops program to use a printer, as well as the screen.
- Think about what it would take to add judging to the Duelling DO Loops program.
- Is it possible to make the Duelling DO Loops program *restart* automatically after each user pattern has been displayed?

LISTINGS

LISTING 5.1 TEXT MAKER

```
*** SYNTAX ERR
>LIST
    5 DIM F$(30),L$(30)
   10 D$="": REM  CONTROL-D
   20 INPUT "FILE NAME===>",F$
   30 GOSUB 500: REM  OPEN THE FILE
   35 INPUT "NEXT LINE? ",L$
   40 IF L$="END" THEN GOSUB 600:
       CALL -936: END
   45 PRINT D$
   50 PRINT D$;"WRITE ";F$
   55 PRINT L$
   60 PRINT D$
   65 GOTO 35
  500 PRINT D$
  505 PRINT D$;"OPEN ";F$
  510 RETURN
  600 PRINT D$
  605 PRINT D$;"CLOSE ";F$
  610 RETURN
```

LISTING 5.2 TEXT LISTER

```
*** SYNTAX ERR
>LIST
    5 DIM F$(30),L$(30)
   10 D$="": REM  CONTROL-D
   20 INPUT "FILE NAME===>",F$
   30 GOSUB 500: REM  OPEN THE FILE
   45 PRINT D$
   50 PRINT D$;"READ ";F$
   55 INPUT L$
   60 PRINT D$
   65 PRINT L$: REM  LIST LINE TO SCRE
      EN
   70 GOTO 45
  500 PRINT D$
  505 PRINT D$;"OPEN ";F$
  510 RETURN
```

LISTING 5.3 MODIFIED TEXT LISTER

```
*** SYNTAX ERR
>LIST
    5 DIM F$(30),L$(30)
   10 D$="": REM  CONTROL-D
   20 INPUT "FILE NAME===>",F$
   30 GOSUB 500: REM  OPEN THE FILE
   45 PRINT D$
   50 PRINT D$;"READ ";F$
   55 INPUT L$
   56 IF L$="EOF" THEN GOTO 99
   60 PRINT D$
   65 PRINT L$: REM  LIST LINE TO SCRE
      EN
   70 GOTO 45
   99 PRINT D$;"CLOSE ";F$
  100 CALL -936: END
  500 PRINT D$
  505 PRINT D$;"OPEN ";F$
  510 RETURN
```

LISTING 5.4 SUBROUTINE LIBRARY LOADER

```
>LIST
    1 REM    ==========================
      ====
    2 REM    =
        =
    3 REM    = SUBROUTINE LIBRARY LOAD
      ER =
    4 REM    =              BY
        =
    5 REM    =  DR. RICHARD C. VILE, J
      R. =
    6 REM    =   ALL COMMERCIAL RIGHTS
        =
    7 REM    =         RESERVED
        =
    8 REM    =
        =
    9 REM    ==========================
      ====
   10 FOR I=1 TO 4: POKE I,0: NEXT
      I
   14 D$="": REM  CONTROL-D
```

LISTING 5.4 *(cont.)*

```
 15 KBD=-16384:CLR=-16368:CLREOL=
    -868:CLREOP=-958:HOME=-936:
    BELL=-198
 16 BANNER=1750:GET=2000
 17 QUIT=4:SHOWSUB=5000
 49 INSTRUCTIONS=10000
 50 DIM SUBR$(30)
100 CALL HOME: POKE 50,255
102 GOSUB BANNER: VTAB 10: TAB
    5
103 PRINT "CHOOSE ONE OF THE FOLLOWI
    NG:"
104 PRINT "(INVERSE LETTERS INDICATE
     SUBS LOADED)"
105 PRINT
106 WHICH=1
110 TAB 5:SUBR$="WAIT": GOSUB SHOWSU
    B
112 TAB 5:SUBR$="GET": GOSUB SHOWSUB

114 TAB 5:SUBR$="DOS COMMANDS":
     GOSUB SHOWSUB
115 TAB 5:SUBR$="EXIT THIS PROGRAM"
    : GOSUB SHOWSUB
195 GOTO 200
199 CALL BELL
200 GOSUB GET
205 TYPE=KEY- ASC("0")
210 IF (TYPE<0) OR (TYPE>QUIT) THEN
    199
215 IF TYPE#QUIT THEN 250
220 GOSUB INSTRUCTIONS: END
250 GOTO 250+TYPE*5
255 REM   ***** WAIT *****
256 PRINT D$;"EXEC WAIT"
257 POKE TYPE,255
259 END
260 REM  ***** GET *****
261 PRINT D$;"EXEC GET"
262 POKE TYPE,255
264 END
265 REM  ***** DOS COMMANDS *****
266 PRINT D$;"EXEC DOS COMMANDS"

267 POKE TYPE,255
269 END
1750 REM       ==================
1751 REM       =      BANNER      =
1752 REM       ==================
1755 VTAB 1: TAB 1: PRINT "=========
     ============================="
     ;
1756 PRINT "=
                    =";
1757 PRINT "=       SUBROUTINE LIBRARY
      LOADER       =";
1758 PRINT "=
                    =";
1759 PRINT "=      BY DR. RICHARD C. V
     ILE, JR.      =";
1760 PRINT "=
                    =";
1761 PRINT "=                  MARCH 198
     1              =";
1762 PRINT "=========================
     ==============";
1799 RETURN
2000 REM ======================
2001 REM = GET KEY SUBROUTINE =
2002 REM ======================
2005 KEY= PEEK (KBD): IF KEY<128
      THEN 2005
2010 POKE CLR,0
2049 RETURN
2100 REM ====================
2101 REM = WAIT SUBROUTINE =
2102 REM ====================
2105 POKE 50,63: VTAB 24: TAB 5:
      PRINT "PRESS ANY KEY TO CONTINU
     E";
2110 POKE CLR,0
2115 GOSUB GET
2120 POKE 50,255
2149 RETURN
5000 REM ====================
5001 REM = SHOW SUBROUTINE =
5002 REM = NAME IN MENU    =
5003 REM ====================
5010 PRINT "(";
5012 IF PEEK (WHICH)=255 THEN POKE
     50,63
5015 PRINT WHICH;: POKE 50,255
5018 PRINT ") ";SUBR$
5040 POKE 50,255
5045 WHICH=WHICH+1
5049 RETURN
10000 REM   ==================
10001 REM   =  INSTRUCTIONS  =
10002 REM   ==================
10005 CALL HOME: VTAB 5: TAB 1
10015 PRINT : PRINT "TO CLEAN UP, ISSU
     E:"
10016 PRINT
10020 PRINT " DEL 0,999"
10025 PRINT " DEL 5000,32767"
10049 RETURN
```

LISTING 5.5 DUELLING DO LOOPS

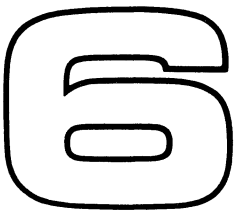
```
>LIST
    1 REM  ============================
    2 REM  =                          =
    3 REM  =   DUELLING DO LOOPS      =
    4 REM  =                          =
    5 REM  =DR. RICHARD C. VILE, JR.=
    6 REM  = ALL COMMERCIAL RIGHTS  =
    7 REM  =       RESERVED           =
    8 REM  =                          =
    9 REM  ============================
   10 DIM A$(40)
   11 D$="": REM   CONTROL-D
   12 KBD=-16384:CLR=-16368
   15 GAME=1000:HOME=-936
   16 WAIT=2100:GET=2000
  112 GAME=1000: REM   CHALLENGE PLAYE
      R
  115 GOSUB GAME
  119 PRINT D$
  120 PRINT D$;"OPEN TEMP"
  125 PRINT D$;"DELETE TEMP"
  130 PRINT D$;"OPEN TEMP"
  135 PRINT "ENTER YOUR SUBROUTINE.  T
      O FINISH,"
  136 PRINT "ENTER '$'"
  137 PRINT
  140 FOR I=500 TO 995 STEP 5
  145 PRINT ">";I;: INPUT A$
  150 IF A$="$" THEN 200
  155 PRINT D$;"WRITE TEMP"
  160 PRINT I;A$
  165 PRINT D$
  170 NEXT I
  200 PRINT D$;"WRITE TEMP"
  202 PRINT I;"RETURN"
  203 PRINT "RUN 250"
  205 PRINT D$;"CLOSE TEMP"
  210 PRINT D$;"EXEC TEMP"
  220 END
  250 CALL -936: GOSUB 500
  300 END
  500 FOR I=1 TO 16
  505 FOR J=1 TO I
  510 PRINT "*";
  515 NEXT J
  520 PRINT
  525 NEXT I
  530 RETURN
 1000 REM  ====================
 1001 REM  = CHALLENGE GAME =
 1002 REM  =                =
 1003 REM  = GENERATE A NEW =
 1004 REM  = PATTERN FOR THE=
 1005 REM  = POOR SLOB TO   =
 1006 REM  = TRY TO MATCH.  =
 1007 REM  =                =
 1008 REM  ====================
 1020 CALL HOME: VTAB 5
 1025 PRINT " I WILL PRINT A PATTERN O
      F ASTERISKS ON"
 1027 PRINT "THE SCREEN.  IT WILL BE Y
      OUR JOB TO "
 1029 PRINT "WRITE A BASIC PROGRAM TO
      DUPLICATE THE"
 1031 PRINT "PATTERN.  YOU MAY USE ONL
      Y FOR STATE-"
 1033 PRINT "MENTS AND THE PRINT STATE
      MENTS:"
 1035 PRINT : PRINT "     PRINT '*';"
 1036 PRINT "     PRINT '*'"
 1037 PRINT "     PRINT ' ';"
 1039 PRINT " AND PRINT "
 1041 PRINT : GOSUB WAIT
 1045 CALL HOME
 1050 P= RND (5)
 1055 H= RND (10)+5:L= RND (10)+5
 1056 S= RND (5)+3
 1060 GOSUB 5000+100*P
 1065 GOSUB WAIT
 1098 CALL HOME
 1099 RETURN
 2000 REM ======================
 2001 REM = GET KEY SUBROUTINE =
 2002 REM ======================
 2005 KEY= PEEK (KBD): IF KEY<128
       THEN 2005
 2010 POKE CLR,0
 2049 RETURN
 2100 REM ==================
 2101 REM = WAIT SUBROUTINE =
 2102 REM ==================
 2105 POKE 50,63: VTAB 24: TAB 5:
       PRINT "PRESS ANY KEY TO CONTINU
      E";
 2110 POKE CLR,0
 2115 GOSUB GET
 2120 POKE 50,255
 2149 RETURN
 5000 REM ===> PATTERN 0 <===
 5005 FOR I=1 TO H
 5010 FOR J=1 TO L
 5015 PRINT "*";
 5025 NEXT J
 5030 PRINT
```

LISTING 5.5 (*cont.*)

```
5035 NEXT I
5099 RETURN
5100 REM ===> PATTERN 1 <===
5105 FOR I=1 TO H
5110 FOR J=1 TO I
5115 PRINT "*";
5120 NEXT J
5125 PRINT
5130 NEXT I
5199 RETURN
5200 REM ===> PATTERN 2 <===
5205 FOR I=1 TO H
5210 FOR J=1 TO H-1
5215 PRINT " ";
5220 NEXT J
5225 FOR J=H-1 TO H
5230 PRINT "*";
5235 NEXT J
5240 PRINT
5245 NEXT I
5299 RETURN
5300 REM ===> PATTERN 3 <===
5305 FOR I=1 TO L-1: PRINT "*";:
     NEXT I
5310 PRINT "*"
5315 FOR I=1 TO H-2
5320 PRINT "*";
5325 FOR J=1 TO L-2
5330 PRINT " ";
5335 NEXT J
5340 PRINT "*"
5345 NEXT I
5350 FOR I=1 TO L-1: PRINT "*";:
     NEXT I
5355 PRINT "*"
5399 RETURN
5400 REM ===> PATTERN 4 <===
5405 FOR I=1 TO H
5410 FOR J=1 TO L
5415 PRINT "*";
5420 NEXT J
5425 FOR K=1 TO S
5430 PRINT " ";
5435 NEXT K
5440 FOR J=1 TO L
5445 PRINT "*";
5450 NEXT J
5455 PRINT
5460 NEXT I
5499 RETURN
5500 REM ===> PATTERN 5 <===
5505 FOR I=1 TO H
5510 FOR J=1 TO H-I+1
5515 PRINT " ";
5520 NEXT J
5525 PRINT "*"
5530 NEXT I
5599 RETURN
5600 REM ===> PATTERN 6 <===
5605 H=10
5610 FOR I=1 TO H
5615 FOR J=1 TO H-I+1
5620 PRINT " ";
5625 NEXT J
5630 PRINT "*";
5635 IF I=1 THEN 5655
5640 FOR K=1 TO 2*I-2
5645 PRINT " ";
5650 NEXT K
5655 PRINT "*"
5660 NEXT I
5699 RETURN
5700 REM ===> PATTERN 7 <===
5705 FOR I=1 TO H
5710 IF I#H/2 THEN 5730
5715 FOR J=1 TO L/2: PRINT "**";
     : NEXT J
5720 PRINT "*"
5725 GOTO 5750
5730 FOR J=1 TO L/2
5735 PRINT " ";
5740 NEXT J
5745 PRINT "*"
5750 NEXT I
5799 RETURN
5800 REM ===> PATTERN 8 <===
5805 FOR I=1 TO H
5810 FOR J=1 TO I
5815 PRINT " ";
5820 NEXT J
5825 FOR J=1 TO I
5830 PRINT "*";
5835 NEXT J
5840 PRINT
5845 NEXT I
5899 RETURN
5900 REM ===> PATTERN 9 <===
5905 FOR I=1 TO H
5910 FOR J=1 TO H-I+1
5915 PRINT " ";
5920 NEXT J
5925 FOR J=1 TO I
5930 PRINT "*";
5935 NEXT J
5940 PRINT
5945 NEXT I
```

```
5950 FOR I=1 TO H
5955 FOR J=1 TO H-I+1
5960 PRINT "*";
5965 NEXT J
5970 PRINT
5975 NEXT I
5999 RETURN

>
```

Chapter



Using Monitor Routines

In this chapter, we shall examine ways in which Integer BASIC programs may take advantage of the machine-language routines located in the Monitor ROM or Autostart ROM. We won't ask you to be a machine-language guru, but working your way through these examples should help to prepare you for our forays into machine and assembly language later on.

1. THE CALL STATEMENT AND PROGRAM PORTABILITY

Integer BASIC allows a programmer to write statements like this:

```
CALL -936
CALL 1
CALL COUT
CALL MOVE
```

In the first two examples, the effect of the statement will be to transfer control from the BASIC program in execution at the time to the machine-language routine located at the RAM memory address indicated by the argument of the CALL statement. (Note: Hexadecimal addresses larger than $8000 must be represented in Integer BASIC as negative decimal integers.) In the last two examples control will pass to the machine-language routines whose addresses are stored in the variable used as the argument to the CALL statement. The following are therefore equivalent as far as Integer BASIC is concerned:

```
HOME = -936
CALL HOME
CALL -936
```

One disadvantage of using CALL statements in your programs is that they become far less *portable*. This means that you cannot expect to enter such a program into a BASIC interpreter on a different computer system and have it work the same way. In fact, in most cases it won't even run at all. This may not seem like a problem to you, since you probably don't have access to more than one computer system anyway. However, you can still appreciate its import. Try running one of your Integer BASIC programs under the APPLESOFT interpreter without making any changes to the program first.

Use CALL identifier
in preference to CALL constant

CALL statements invoke machine-language routines which are located at a particular address. When writing a CALL statement in a BASIC program, it is a good idea to use the form

CALL Identifier

where a suitable identifier has been chosen to represent the machine-language routine and assigned as its value the address of that routine. The use of the identifier in preference to the constant makes the code much easier to understand. Some people seem to have a natural recall for numbers and no trouble remembering the meanings of CALL −936, CALL −198, CALL −868, etc. On the other hand, CALL HOME, CALL BELL, CALL CLREOL, etc., are just as easy to write and far gentler on the reader whose natural bent is not toward memorizing numbers.

The use of CALL Identifier should be coupled with an appropriate assignment to Identifier, namely the address of the routine to be invoked. This assignment should be located in the declaration section of the program (see Chapter 2). An additional advantage of this approach is that you may substitute your own machine-language routines for the Monitor routines. This may be accomplished by simply changing the assignment in the declaration section to "point" to your routine. Then all references to CALL Identifier for that particular identifier will invoke your code and not the Monitor's.

Machine-Language Routine Input Parameters

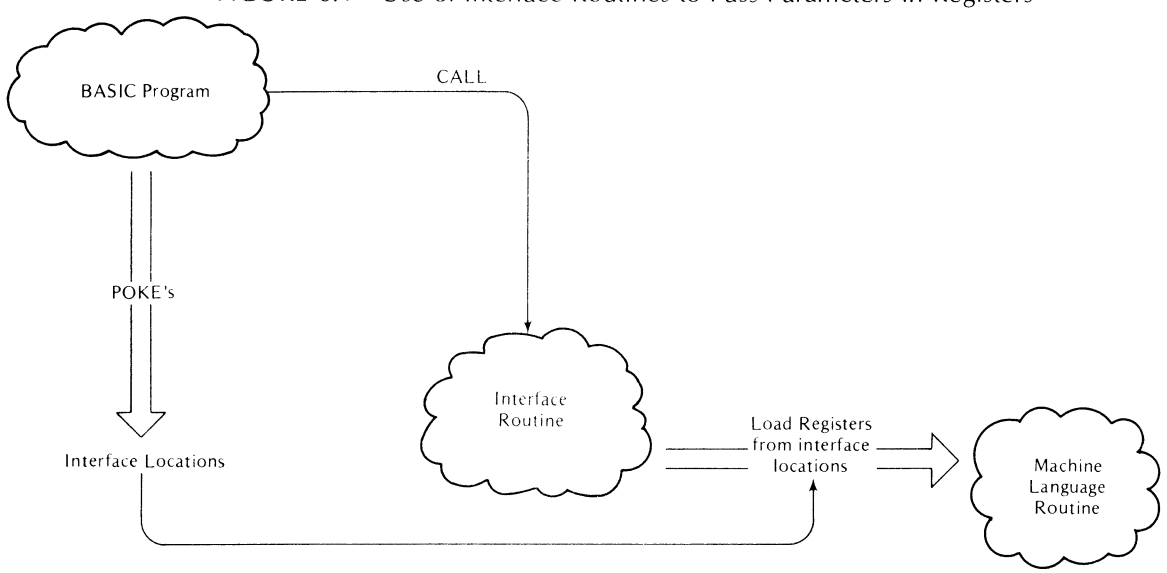
Many machine-language routines require certain input values to be provided by the caller. There are basically two methods by which this may be accomplished:

- Input parameters may be passed to the routine in registers: accumulator, X register, Y register.
- Input parameters may be passed to the routine in memory locations. In particular, page zero locations (addresses 0–255) are frequently used for this purpose.

Routines which use the second method are easier to CALL from BASIC. The BASIC program issuing the CALL simply uses POKE statements to place the inputs to the CALLed routine into the appropriate memory locations. For routines which use the first method, a bit more trickery is necessary. This is illustrated in Figure 6.1. A small "interface" routine must be written whose purpose is to set up the appropriate machine registers with the information required by the target machine-language routine. These values are obtained by the interface routine from certain "agreed upon" memory locations, denoted in Figure 6.1 as "Interface Locations." The BASIC program will use POKE statements to place values into the interface locations before calling the interface routine.

We shall give examples of both methods of passing information. One example will involve the Monitor MOVE and VTAB routines and the other will illustrate the use of COUT.

FIGURE 6.1 Use of Interface Routines to Pass Parameters in Registers



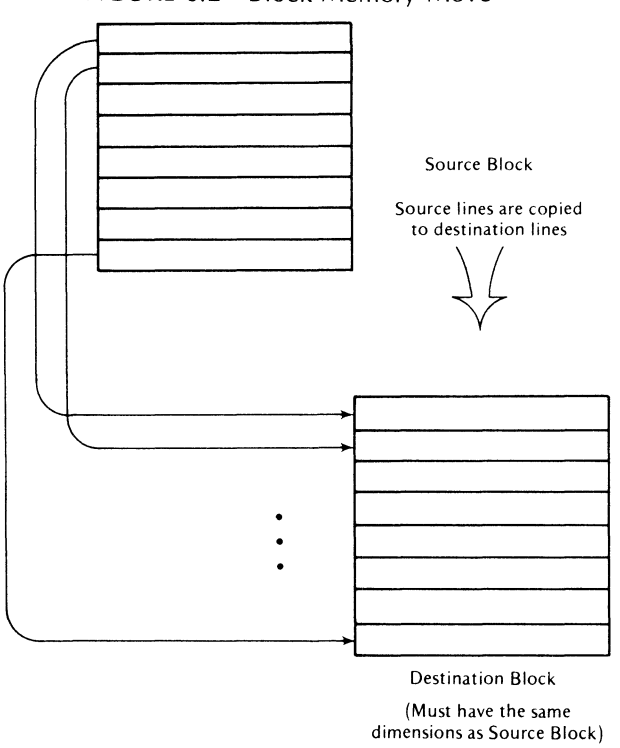
2. THE WINDOW SHADES PROGRAM

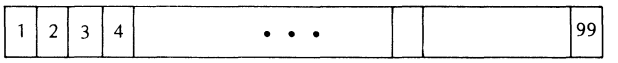
The program of Listing 6.1 uses routines in the Monitor (or Autostart) ROM to implement yet another dynamic low-resolution graphics display. The basic idea of the program is that blocks within the low-resolution screen are copied from one place to another on the screen. This will gradually transform the original design on the screen and is particularly interesting when the source and destination blocks overlap.

Figure 6.2 schematically illustrates the program's operation. A block contained within the graphics screen will be treated as if it were a text window (see Chapter 2). It is divided into a number of lines, each of which actually accounts for two rows within the graphics display (recall that there are 24 text lines on the APPLE II screen, but 48 graphics rows). Each line in the source block will be copied onto the corresponding line in the destination block. Figure 6.2 indicates this process with arrows connecting the respective lines. Sometimes this copying will proceed from the top to the bottom of each block and other times it will reverse direction and go from the bottom to the top of each block. This up and down motion suggested the title of Window Shades for the program.

To gain a feeling for what may happen when such a process is carried out, let's consider an analog in one dimension. Suppose you were to copy the contents of a

FIGURE 6.2 Block Memory Move





Copy array positions 1 – 50 to positions 4 – 53

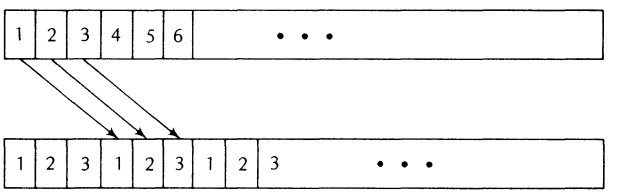


FIGURE 6.3 Overlapped Copy of an Array

simple array of memory locations from one place to another in computer memory. If the copy were made into a totally separate array of memory locations containing no locations in common with the original array, then all the information in the original array would be preserved identically in the copy. On the other hand, if the array were copied into another array which overlapped the original in some fashion, then some of the original information would be lost. It would get written on top of by other information as part of the copy process. This is illustrated in Figure 6.3.

In this example, we begin with an array of 100 locations containing the consecutive integers from 1 to 100. If array positions 1 to 50 are copied (in order) into positions 4–53 in the same array, the result is a repeating sequence of the pattern 1, 2, 3, 1, 2, 3, ... The reason is that when it comes time to copy array entry number 4, the original value of 4 has been replaced (overwritten and destroyed) by the value 1 due to the early part of the copy. Likewise 5 gets replaced by 2, 6 by 3, 7 by 1, and so on.

The result of the overlapping copy is a strongly repetitive pattern. The length of the repetition is determined by the size of the nonoverlapping part of the copy. This technique is sometimes used to set all values of an array to a given value. Think for example what would happen if array positions 1–98 were copied to positions 2–99.

In the two-dimensional case of the Window Shades program, interesting repetitive graphics patterns are produced by the block-copying process. Now let us turn to the implementation of the program.

Window Shades Implemented

The Window Shades program will make use of the Monitor MOVE routine to copy one block of screen to another. The MOVE routine is not limited to copying portions of screen memory, but may transfer any consecutive set of

locations from one place to another in memory. The MOVE routine expects three pieces of information:

1. The address in memory to begin copying *from*.
2. The address in memory to begin copying *to*.
3. The address in memory at which to *stop* copying.

It expects this information to be provided in certain memory locations in page zero:

1. Locations 60,61.
2. Locations 66,67.
3. Locations 62,63.

Notice that each piece of information requires two bytes rather than one. This is because the address of an APPLE II memory location may in general require 16 bits or two bytes to represent it.

The three addresses required by the MOVE routine are provided by the VTAB routine. VTAB is another subroutine that forms part of the Monitor and Autostart ROMs. It uses two pieces of information in order to calculate a memory location that corresponds to a desired position on the screen. The first of these is simply a number between 0 and 23 which represents a line number on the screen (screen line numbers are an example of 0-indexing; see Chapter 4). Normally this is used to change the vertical position of the cursor on the text screen. We are using it to calculate addresses to pass on to the MOVE routine. The other piece of information used by VTAB is a number which represents where within a given line the leftmost position of the text window resides. This is used in order to keep the cursor inside the text window during vertical positioning.

Like MOVE, VTAB expects to find its information in certain page zero locations. In particular, it wants CV (cursor vertical position) to be in location 37 and WNDLFT (WiNDow LeFT index) to be in location 32. Of course, both of these pieces of information fit easily into a single byte.

When VTAB is called, it in turn calls a routine named BASCALC. BASCALC calculates the address of the memory location which corresponds to the beginning of a given screen line: namely, the line whose number is indicated by the contents of the accumulator when BASCALC is entered. VTAB provides this by placing the contents of CV into the accumulator before calling BASCALC. BASCALC stores the result of its calculation into the two page zero locations BASL and BASH (locations 40 and 41). Upon return from BASCALC, VTAB adds in the value of WNDLFT in order to give the address of the memory location corresponding to the first screen position on the line which falls within the window. All of this furious activity is illustrated in Figure 6.4.

The BASIC portion of the Window Shades program

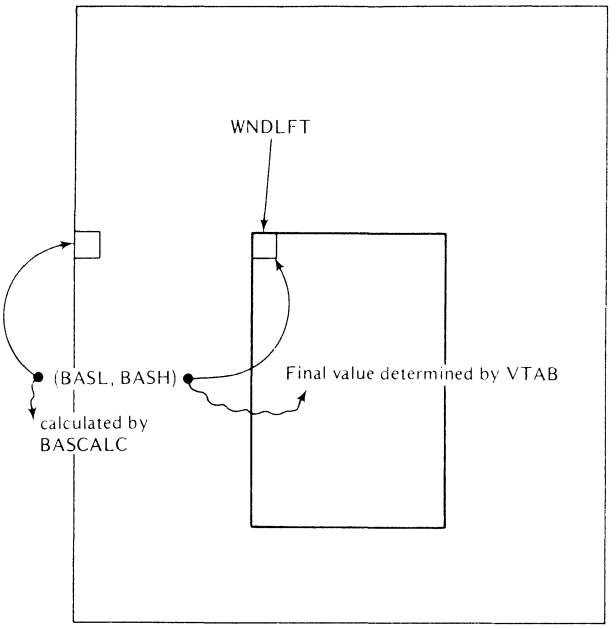
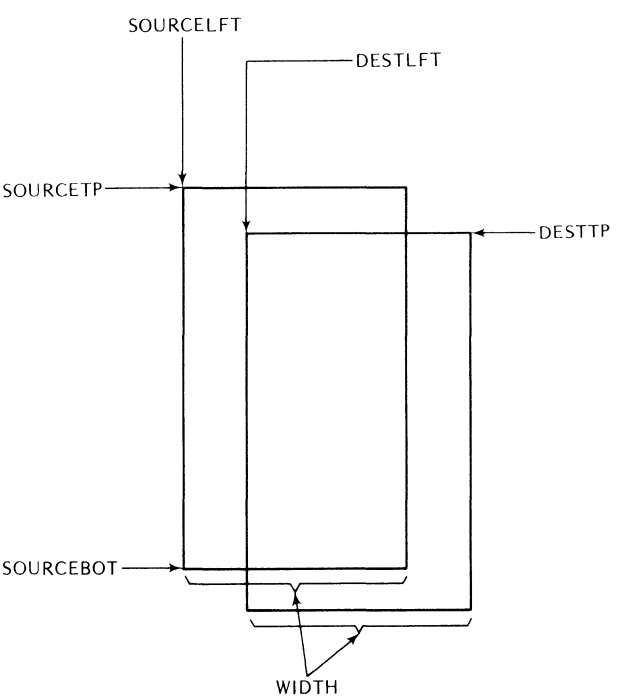


FIGURE 6.4 VTAB's Activities

must provide MOVE with its three inputs. Figure 6.5 introduces some notation that is used by the program while accomplishing this. The copy is made a line at a time, and MOVE is called each time. Using the discussion already presented, see if you can figure out how the

FIGURE 6.5 Notation for Window Shades Copy Process



communication with COUT is accomplished. Program lines 510 to 650 contain the relevant statements.

The parameters used are selected randomly in lines 100 to 140. The initial display is generated using code similar to that of the VIDEO program of Chapter 3.

3. USING INTERFACE ROUTINES

The Monitor routine COUT is an example of a machine-language subroutine that expects to find its input parameter located in a machine register. COUT in fact takes the contents of the 6502 accumulator and outputs it to the current output device. For our purposes, we may think of COUT as strictly using the screen. The character that COUT puts on the screen is the character whose code appears in the accumulator when COUT is entered.

When BASIC executes a PRINT statement with a string of characters as its argument, such as PRINT "HELLO", it dissects the string a character at a time and feeds the individual characters to COUT. The characters in the string are, of course, what the programmer keyed in as the string part of the PRINT statement when the program was originally written. This brings up a slight sore point with the APPLE II:

> There are characters which may be displayed on an APPLE II screen which cannot be produced by direct input from the keyboard.

APPLE II programmers have invented a variety of schemes to get around this annoyance. One of them involves the use of the COUT routine, and this we shall now demonstrate.

If we could only find a way to force feed COUT with the desired character value (in the accumulator), then we could produce just about anything we wanted to on the display. At least we would not be limited to what we could key in as part of a PRINT statement. Unfortunately, there is no Integer BASIC command to "Load the accumulator."

The secret lies in writing a simple machine-language interface routine which does the force feeding for us, using a value which we obligingly POKE into memory first. The routine is shown in Listing 6.2 and as you can see it is extremely simple. All it does is to pluck a number out of memory, load it into the accumulator, and CALL the COUT subroutine. Thus, we accomplish by devious indirect means what we cannot do directly.

The uses of this technique are a little limited. One example is shown in Listing 6.3. This program produces a menu in which the choice letters are enclosed in square brackets. Since the character "[" cannot be produced from the keyboard in Integer BASIC, it is printed using the COUT interface of Listing 6.2.

Notice the technique for loading the machine-language routine itself. The code is POKEd into memory at some available page zero locations as part of the program's initialization process. In general, locations 2–31 are safe locations for activities of this nature. Locations 0 and 1 are used by the Monitor only for temporary storage. Thus they are safe for passing values to interface routines, but probably should not be used for storing code.

4. EXPLORATIONS

- Modify the Window Shades copy algorithm to use other orders besides top-to-bottom or bottom-to-top. For example, you could copy even numbered lines from top-to-bottom and odd numbered lines from bottom-to-top.
- Modify the Window Shades program to use the text display instead of the graphics display. In a way, this is a more surprising program than the graphics version, at least the first time you watch it.
- Experiment with other methods of producing the background portion of the Window Shades display. A possible source of techniques may be found in Chapter 3.
- Experiment with other limits for the sizes of the source and destination blocks in the Window Shades program. Notice that the program of Listing 6.1 uses a very conservative approach in order to guarantee easily that the destination block does not "slop off" the screen. Add more code to both allow larger blocks and still make sure that there are no overflow problems.
- The program of Listing 6.1 always moves the source block down and to the right. Modify the program to allow other possibilities for the relationship between the source and destination blocks.
- Investigate other Monitor routines that require parameters to be passed in registers. See if you can write interface routines to make them invocable from Integer BASIC.

LISTINGS

LISTING 6.1 WINDOW SHADES

```
>LIST
    1 REM ============================
    2 REM =                            =
    3 REM =       WINDOW SHADES        =
    4 REM =        WRITTEN BY          =
    5 REM = DR. RICHARD C. VILE, JR.   =
    6 REM =  ALL COMMERCIAL RIGHTS     =
    7 REM =        RESERVED            =
    8 REM =                            =
    9 REM ============================

   10 KBD=-16384:CLR=-16368
   11 DISPLAY=400:DRAWSHADES=500
   12 WAIT=700:HOME=-936
   13 FULL=-16302:CV=37:WNDLFT=32
      :VTAB=-990:MOVE=-468
   90 GOSUB DISPLAY
  100 SOURCETP= RND (6)
  105 SOURCEBOT=17+ RND (6)
  110 WIDTH=10+ RND (20)
  120 SOURCELFT= RND (10)
  130 DESTTP=SOURCETP+2
  140 DESTLFT=SOURCELFT+ RND (5)
  150 GOSUB DRAWSHADES
  155 RC=0: GOSUB WAIT
  160 IF RC#0 THEN GOTO 90
  170 GOTO 100
  400 REM ========================
  401 REM =     D I S P L A Y      =
  402 REM =                        =
  403 REM =   GENERATE AN INITIAL  =
  404 REM = GRAPHICS DISPLAY FOR   =
  405 REM = THE PROGRAM TO MODIFY. =
  406 REM = ANY PATTERN WILL DO -  =
  407 REM = WE USE A VERSION OF    =
  408 REM = OF THE "VIDEO" PRO-    =
  409 REM = GRAM PRESENTED IN      =
  410 REM = CHAPTER 3.             =
  411 REM ========================
  420 GR : POKE FULL,0: COLOR=0
  425 FOR LINE=40 TO 47: HLIN 0,39
       AT LINE: NEXT LINE
  430 COLOR= RND (16)+1
  435 FOR LINE=0 TO 19+ RND (20)
```

LISTING 6.1 *(cont.)*

```
440 HLIN 0,39 AT LINE: HLIN 0,39
     AT 47-LINE
450 IF RND (2)=0 THEN COLOR= RND
    (16)+1
460 VLIN 0,47 AT LINE: VLIN 0,47
     AT 39-LINE
465 NEXT LINE
499 RETURN
500 REM ============================
501 REM =    D R A W S H A D E S    =
502 REM =                           =
503 REM =    CALL MONITOR MOVE      =
504 REM = ROUTINE TO SIMULATE UP    =
505 REM = AND DOWN WINDOW SHADES.   =
506 REM ============================
508 WHICH= RND (2): IF WHICH=1 THEN
    515
510 FOR LINE=SOURCETP TO SOURCEBOT
511 GOTO 520
515 FOR LINE=SOURCEBOT TO SOURCETP STEP -1
520 POKE CV,LINE
530 POKE WNDLFT,SOURCELFT
540 CALL VTAB
550 POKE 60, PEEK (40)
560 POKE 61, PEEK (41)
570 POKE 62, PEEK (40)+WIDTH
580 POKE 63, PEEK (41)
590 POKE CV,DESTTP+(LINE-SOURCETP)

600 POKE WNDLFT,DESTLFT
610 CALL VTAB
620 POKE 66, PEEK (40)
630 POKE 67, PEEK (41)
640 CALL MOVE
650 NEXT LINE
660 RETURN
700 REM ====================
701 REM =    W  A  I  T     =
702 REM ====================
705 KEY= PEEK (KBD): IF KEY>=128
     THEN 710
706 RC=0: RETURN
710 POKE CLR,0: IF KEY# ASC("Q"
    ) THEN 720
715 TEXT : CALL HOME: END
720 RC=1
749 RETURN
```

LISTING 6.2 COUT INTERFACE

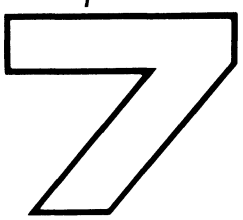
```
0000:                2 ;*
0000:                3 ;*
0000:                4 ;**********************************
0000:                5 ;*                                *
0000:                6 ;*   COUT INTERFACE ROUTINE       *
0000:                7 ;*                                *
0000:                8 ;* PICK UP THE VALUE IN LOC. 0    *
0000:                9 ;* PUT IT INTO THE AC AND CALL    *
0000:               10 ;* COUT.                          *
0000:               11 ;*                                *
0000:               12 ;**********************************
0000:               13 ;*
0000:               14 ;*
----- NEXT OBJECT FILE NAME IS LISTING 6.2 *****.OBJ0
0002:               15          ORG   $02
FDED:               16 COUT     EQU   $FDED
0002:               17          ENTRY START
0002:               18 ;*
0002:               19 ;*
0002:A5 00          20 START    LDA   $00
0004:20 ED FD       21          JSR   COUT
0007:60             22          RTS

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

LISTING 6.3 COUT MENU EXAMPLE

```
>LIST
   10 COUTPOKES=1000
   11 HOME=-936
   40 GOSUB COUTPOKES
   50 CALL HOME
   54 POKE 0,219
   55 VTAB 5: TAB 5: PRINT "CHOOSE ONE OF THE FOLLOWING:": PRINT
   60 TAB 5: CALL 2: PRINT "A]  ICE CREAM"
   61 TAB 5: CALL 2: PRINT "B]  HOT DOGS"
   62 TAB 5: CALL 2: PRINT "C]  HAMBURGER"
   63 TAB 5: CALL 2: PRINT "C]  FILET MIGNON"
   64 TAB 5: CALL 2: PRINT "D]  SHRIMP COCKTAIL"
   65 TAB 5: CALL 2: PRINT "E]  BEEF WELLINGTON"
   66 TAB 5: CALL 2: PRINT "F]  COQUILLES DE SAINT JACQUES"
   67 TAB 5: CALL 2: PRINT "G]  RIJSTAFFEL"
   68 TAB 5: CALL 2: PRINT "H]  BROILED SEA BASS"
   69 TAB 5: CALL 2: PRINT "J]  NEW ENGLAND CLAM CHOWDER"
   70 TAB 5: CALL 2: PRINT "K]  SEVICHE"
```

```
  71 TAB 5: CALL 2: PRINT "L]  GAZPACHO"
  99 END
1000 REM ====================
1001 REM = POKE IN THE COUT =
1002 REM = INTERFACE ROUTINE=
1003 REM ====================
1010 POKE 2,165
1011 POKE 3,0
1012 POKE 4,32
1013 POKE 5,237
1014 POKE 6,253
1015 POKE 7,96
1049 RETURN

>
```

Chapter



APPLESOFT BASIC: Features and Use

APPLESOFT BASIC is a stripped down version of Microsoft BASIC. It is the "other" BASIC interpreter for the APPLE. We shall give a very brief review of its characteristics and compare and contrast it with Integer BASIC.

Introducing APPLESOFT

APPLESOFT BASIC is much more "standard" than Integer BASIC. It compares closely with the more popular style of BASIC. In particular, APPLESOFT features:

- Floating-point variables
- Strings and string arrays
- Short variable names
- String functions: RIGHT$, LEFT$, MID$
- Scientific functions: COS, TAN, LOG, etc.
- DATA and READ statements

Most of these features are not available in or clash with similar features in Integer BASIC. For example, Integer BASIC does not have string arrays and lacks the RIGHT$, LEFT$, and MID$ functions.

Fun and Games

APPLESOFT provides statements to give access to the APPLE II's entertainment features: graphics, game paddles, random numbers, etc. These statements may be used in much the same way as the corresponding Integer BASIC statements.

The use of the RND statement is different. Since APPLESOFT is oriented toward floating-point numbers, the RND function produces a floating-point value. In fact:

```
0 <= RND(expr) < 1.0
```

In order to generate integer values, the following sort of expressions must be used:

```
(1)  RA = INT(RND(EXPRESSION)*N)
(2)  RA = INT(RND(EXPRESSION)*N + 1)
```

Statement (1) produces a value between 0 and N − 1, inclusive. Statement (2) produces a value between 1 and N, inclusive.

Display Features

APPLESOFT gives the use of VTAB and HTAB statements for positioning the cursor on the text display. It also supports the scrolling window via the same page zero locations as described in Chapter 2. Many of the programming techniques utilizing the APPLE display are identical or very similar to those already presented for Integer BASIC. In Chapter 10, we present an APPLESOFT version of the APPLE Trivia Quiz which demonstrates such techniques.

Graphics

APPLESOFT supports the same statements for accessing the low-resolution graphics features that Integer BASIC does. Again, many of the techniques explained in Chapter 3 for Integer BASIC apply directly to APPLESOFT.

In addition APPLESOFT provides support for the APPLE II high-resolution graphics. It introduces the concept of a shape table and allows the user to command the interpreter to DRAW shapes from the table. It also provides statements to select either of the high-resolution display pages for viewing (HGR, HGR2), and statements for plotting lines and points (HPLOT, HPLOT TO, etc.).

The topic of APPLESOFT high-resolution graphics fits into the much larger topic of APPLE high-resolution graphics in general. We keep our emphasis in this volume on low-resolution graphics and reserve coverage of high-resolution topics for a future work.

Use of APPLE DOS

APPLESOFT interacts with the APPLE Disk Operating System in much the same way that Integer BASIC does. (Is this beginning to sound like a broken record?) The concepts introduced in Chapter 5 may be applied in an APPLESOFT setting. We give an example of such in Chapter 8 where we implement a very simple text file editor.

Arrays and Strings

APPLESOFT goes beyond Integer BASIC in its array and string-handling capabilities. It allows not only arrays of numbers, but arrays of strings as well. It provides the user with arrays of more than a single dimension. It allows the user to manipulate strings with built-in functions that extract substrings and a concatenation operator that builds up strings.

In Chapters 9 and 10, we explore the use of both arrays and string manipulation in APPLESOFT.

When to Use APPLESOFT

Since the APPLESOFT interpreter supports floating-point arithmetic, as well as many features not found in Integer BASIC, its use is preferable in many situations. The main disadvantages of using APPLESOFT are as follows:

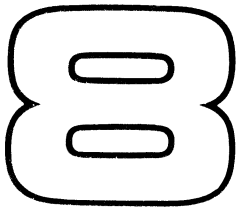
- The interpreter is less efficient, due to its greater generality.
- APPLESOFT programs are harder to read. They do not give the user as many self-documenting features as do Integer BASIC programs.

The advantages of using APPLESOFT are as follows:

- The availability of far more features.
- The use of a more portable version of BASIC.

In general, if blinding speed is not the objective and if programs are likely to be moved to different computers, it is preferable to use APPLESOFT.

Chapter



APPLE DOS and APPLESOFT

In this chapter, we discuss usage of DOS within APPLESOFT programs. Most of the concepts are the same as with the use of DOS within Integer BASIC programs, so a review of Chapter 5 might be in order before beginning.

1. COMPARISON OF USAGE

There are very few differences in usage of DOS when coding in APPLESOFT as compared to coding in Integer BASIC. If you have mastered the concepts of Chapter 5, you should be able to begin writing APPLESOFT programs which use files immediately.

Control-D

APPLESOFT provides the CHR$ function, which Integer BASIC lacks. This allows the use of the statement:

```
D$ = CHR$(4)
```

to define Control-D, instead of:

```
D$ = CHR$(4):REM CONTROL-D
```

with its "invisible" Control-D.

Filenames

Integer BASIC strings are really character arrays. This means that a DIM statement must be used to "declare" each individual filename used by a program, e.g.:

```
DIM INFILE$(30),OUTFILE$(30)
```

In APPLESOFT, a simple string variable may be used to hold a filename. The DIM statements are inappropriate and superfluous, unless of course you wish to declare an array of strings to store *several* filenames. For an example of this usage, see Chapter 9.

2. ERROR HANDLING

APPLESOFT BASIC provides the user with an ONERR GOTO statement, which causes transfer of control to a specified line number when an error occurs. This provides an alternate method of detecting "end of file" or "out of data" than the one suggested in Chapter 5. More often than not when writing a program that reads and processes data from a file, you don't know when to expect the end of

that data. In Chapter 5, we advocated the use of a symbolic end of file to get around this problem. The APPLESOFT ONERR GOTO statement provides an alternate method of handling it. Even with this statement, however, the use of a symbolic end of file is sometimes preferable. We shall comment on this further later on.

3. SIMPLE PROGRAMS

The programs of Listings 8.1 and 8.2 at the end of the chapter may be used to create and list text files. They are included to allow comparison with similar programs presented in Chapter 5. Notice the use of the ONERR GOTO statement to catch the end of the file in the second program.

4. APPLICATION: A SIMPLE TEXT EDITOR

The program of Listing 8.3 presents an extremely simple editor for DOS textfiles. It makes provision for a single input file and a single output file. All editing takes place in a single sequential pass through the input file. Figure 8.1 shows the general flow of an edit session.

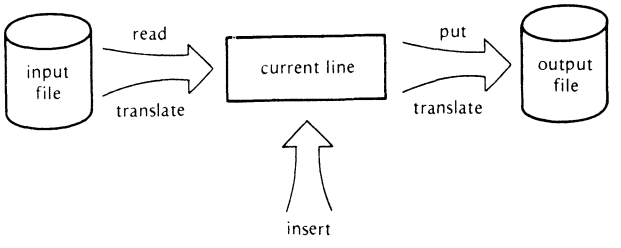
The editor makes use of the concept of a "current line." At any point during an edit session, the editor maintains a string known as the current line. Initially this line is empty, but it changes as the result of executing various editor commands. The variable LI$ maintains the current line.

Sampling the keyboard in APPLESOFT

The editor accepts single letter commands, which it retrieves by use of the GET statement:

```
GET KE$
```

FIGURE 8.1 APPLESOFT Editor Actions



This returns the key struck by the user, including control keys. Thus, the program may not be interrupted by typing a Control-C in response to a GET statement. Other than that, GET behaves much like the GET subroutine used in many of the Integer BASIC programs earlier.

The single letter commands are fielded by the editor, which prompts the user for any additional information required in order to carry out the command. The following commands have been implemented in this version of the editor:

O Open Files

Prompts are issued for input and output filenames, and the corresponding files are opened. The names of the input and output files are stored, respectively, in the string variables FI$ and FO$.

C Close Files

The remainder of the input file is transferred to the output file (automatically) and both files are then closed. Since end of data is being trapped by the use of an ONERR GOTO statement, it doesn't matter if there is actually any input file to read from or not. This means that a file may be created by simply adding lines to the output file, while using a nonexistent input file. The input file will be created as a side effect and it may be desirable to delete it after the edit session.

The automatic transfer of lines is done so that the user does not have to perform this function manually after editing a few lines at the beginning of a file.

When the files are closed, both file names are "erased." That is, FI$="" and FO$="". If this is not done, then an inadvertent READ command will cause the input file to be reopened.

R Read a Line

The next line of the input file is read and made the current line. It is also displayed on the screen for the user's consideration.

Note: if the input file is empty, doing a READ will cause both it and the output file to be closed immediately.

P Put a Line

If the current line is not an empty line, it is written to the output file. The current line remains the same after the operation. Therefore, two PUT commands in a row results in two copies of the current line in the output file.

I Insert Lines

This command causes the editor to go into *insert mode*. This allows the user to insert as many lines as desired into the output file. The lines are placed after the line referenced by the last P, I, or T command. The input mode may be terminated by typing a line equal to $.

T Transfer Lines

The editor goes into *transfer mode*. This causes lines to be read from the input file and allows the user to either pass them on to the output file, or to skip over them. The transfer mode is terminated by simply hitting the ESC key.

S Show Current Line

The current line is displayed on the screen. This allows the user to find out what the current line actually is in case of confusion.

L List Input File

Causes the remainder of the input file to be listed on the screen. Normally used as a standalone command when it is desired to simply list the entire contents of a text file.

Note: Both input and output files will be closed after executing this command.

E End the Edit Session

Clears the screen and returns to APPLESOFT. If files are open, it first executes a CLOSE operation.

? DOS Command

The command causes a ? prompt to appear on the screen and the string read in response to be issued as a DOS command. This allows the user to view the catalog from the editor. It also allows junk files to be DELETEd or RENAMEd.

H Help: Explain other commands

Brief explanations are given of the workings of the Editor's commands. The user is prompted

WHICH COMMAND?

and then types a letter, just as if issuing the actual command itself.

Write user-friendly programs

I'm not sure who coined the phrase "user-friendly," but it is an accepted buzz phrase in professional software development circles. It has no hard and fast definition—it is more an expression of a philosophy than a technical term. It really focuses attention on the ease of use and flexibility of a program. It is an ideal to be striven toward, but never taken for granted.

One technique for user-friendly software is:

- Never assume the user won't make a mistake.

This statement covers a multitude of sins. As an example of what is intended, consider the command decoding portion of the APPLESOFT Editor. The ON GOTO statement in line 210 covers values of TY from 1 to 26; each value corresponds to one of the letters of the alphabet. For those letters which *do not* correspond to an editor command, control is passed to line 250, where the editor beeps and prints the message:

NOT A COMMAND. PLEASE TRY AGAIN.

This is a slightly more polite way to inform the user that: "I don't know that command, dummy! Stick to the script!"

The existence of line 250 is not absolutely necessitated by the nature of the ON GOTO statement. For example, we could have just used line 200 as the destination of the GOTO in those cases. But where would that leave the user?

→ In the dark!

Nothing visible would happen except that the editor would redisplay its prompt character with no message. Now the user might *assume* that an illegal command had been entered. On the other hand, the user might panic, thinking that the editor had taken drastic action, such as deleting the input file. A really paranoid user might freeze up and refuse to go any further with the stupid editor.

Printing the helpful message doesn't hurt the editor and it *may* help the user, so why not do it? Likewise, consider line 209:

209 IF TY <= 0 OR TY > 26 THEN 250

The line *could have been* left out of the program with no harm to the flow of control. The editor would still run, and illegal commands would simply fall through to the GOTO statement in line 210. But again, it is a courtesy to the user to include this helpful protection.

This example may seem picayune and perhaps in its context, it is. But, it illustrates a particular *approach* to things—a general spirit or attitude if you will. It says that a program should provide all the information possible and practical to the user, however minute. In the process, the program probably will wind up protecting *itself* from one or more disasters. Making this approach a habit will pay you big dividends when you start writing more complex programs.

Another technique for user-friendly software is:

- Provide on-line help facilities

The Help command in the APPLESOFT editor is a good example of this. It is not necessary for the user to have a manual at hand in order to use the editor. Simply typing H followed by a command letter will provide a brief explanation of that command. This may seem like a lot of extra work in such a simple program, but users really appreciate things like that.

5. EXPLORATIONS

• Add to the Editor Help feature. Allow the user to request information about the "state" of the edit. For example, what are the current input and output files?

• Can you extend the command set of the editor without taking up more display space for the command list prompts? How about allowing the user to "toggle" between two sets of command prompt displays.

• Investigate the limitations of APPLESOFT input which do not allow commas to be present in certain circumstances. How does this affect the current editor program?

LISTINGS

LISTING 8.1 TEXT FILE CREATOR

```
]LIST

10 D$ =  CHR$ (4)
20  INPUT "FILE NAME===>";F$
30  GOSUB 500: REM  OPEN THE FILE

35  INPUT "NEXT LINE";L$
40  IF L$ = "END" THEN  GOSUB 600
     : HOME : END
45  PRINT D$
50  PRINT D$;"WRITE ";F$
55  PRINT L$
60  PRINT D$
65  GOTO 35
500  PRINT D$
505  PRINT D$;"OPEN ";F$
510  RETURN
600  PRINT D$
605  PRINT D$;"CLOSE ";F$
610  RETURN

]
```

LISTING 8.2 TEXT FILE LISTER

```
]LIST

10 D$ =  CHR$ (4)
15  ONERR  GOTO 99
20  INPUT "FILE NAME===>";F$
30  GOSUB 500: REM  OPEN THE FILE

40  PRINT D$
50  PRINT D$;"READ ";F$
55  INPUT L$
60  PRINT D$
65  PRINT L$: REM  LIST LINE TO S
     CREEN
70  GOTO 40
99  PRINT D$
100  PRINT D$;"CLOSE ";F$
101  INVERSE : PRINT "END OF FILE
     - HIT ENTER TO CLEAR SCREEN
     "
102  GET A$
103  NORMAL : HOME : END
500  PRINT D$
505  PRINT D$;"OPEN ";F$
510  RETURN

]
```

LISTING 8.3 APPLESOFT TEXT EDITOR

```
]LIST

1   REM ============================

2   REM =                          =

3   REM = APPLESOFT FILE EDITOR    =

4   REM =            BY            =

5   REM =DR. RICHARD C. VILE, JR.=

6   REM = ALL COMMERCIAL RIGHTS    =

7   REM =        RESERVED          =

8   REM =                          =

9   REM ============================
15 KB =  - 16384:CL =  - 16368
20 G$ =  CHR$ (7): REM  BELL
21 D$ =  CHR$ (4): REM  DOS WARNI
    NG CHARACTER
80  HOME : HTAB 15: INVERSE : PRINT
    "COMMANDS"
81  NORMAL : PRINT "R - READ LINE
    ";: HTAB 20: PRINT "D - DELE
    TE LINE"
82  PRINT "I - INSERT LINE";: HTAB
    20: PRINT "O - OPEN INPUT FI
    LE"
83  PRINT "P - PUT LINE";: HTAB 2
    0: PRINT "C - CLOSE OUTPUT F
    ILE";
84  PRINT "T - TRANSFER LINES";: HTAB
    20: PRINT "S - SHOW CURRENT
    LINE";
85  PRINT "E - EXIT PROGRAM";: HTAB
```

LISTING 8.3 (*cont.*)

```
     20: PRINT "L - LIST INPUT FI
     LE"
86   PRINT "H - HELP";: HTAB 20: PRINT
     "? - DOS COMMAND"
89   PRINT "!---------!---------!-
     ---------!---------";
90   GOSUB 30000: REM  SET UP WIND
     OW
93   PRINT D$: PRINT D$;"CLOSE ";F
     O$
98 T = 0:EF = 0
99   ONERR  GOTO 1090
200   PRINT "*";: GET KE$
201   IF  ASC (KE$) <  > ASC ("?"
     ) THEN 207
202   INPUT CM$: IF CM$ = "CATALOG
     " THEN  TEXT : HOME : PRINT
     D$;CM$: GOTO 80
203   PRINT D$;CM$
207   PRINT KE$;"      ";
208 TY =  ASC (KE$) -  ASC ("@")
209   IF TY <  = 0 OR TY > 26 THEN
     250
210   ON TY GOTO 250,250,1000,1100
     ,1200,250,250,1300,1400,250,
     250,1150,250,250,1500,1600,2
     50,1700,1900,1800,250,250,25
     0,250,250,250
215   GOTO 200
250   PRINT G$;"UNKNOWN COMMAND": GOTO
     200
1000   REM ====================
1001   REM = CLOSE OUTPUT FILE =
1002   REM ====================
1015   IF LI$ <  > "" THEN 1050
1016   IF EF = 1 THEN 1092
1020   PRINT D$: PRINT D$;"READ ";
     FI$
1025   INPUT LI$
1050   PRINT D$: PRINT D$;"WRITE "
     ;FO$
1055   PRINT LI$
1060   GOTO 1020
1090   IF  PEEK (222) <  > 5 THEN
       PRINT "FATAL ERROR NUMBER "
     ; PEEK (222): TEXT : END
1091   PRINT "====================
     ========>END OF FILE";:EF =
     1
1092   INPUT "DO YOU WISH TO CLOSE
     (Y/N)";A$: IF A$ <  > "Y" THEN
     1098
1093   PRINT D$: PRINT D$;"CLOSE "
     ;FO$
1094   PRINT D$: PRINT D$;"CLOSE "
     ;FI$
1095 FI$ = "":FO$ = "":EF = 0
1098 T = 0
1099   GOTO 200
1100   REM ======================
     ====
1101   REM = DELETE THE CURRENT LI
     NE =
1102   REM ======================
     ====
1105   PRINT "DELETE CURRENT LINE"

1110 LI$ = ""
1149   GOTO 200
1150   REM ==================
1151   REM = LIST INPUT FILE =
1152   REM ==================
1155 T = 1
1156   HOME
1160   FOR I = 1 TO 10: GOSUB 1700
     : NEXT I
1165   PRINT "TO CONTINUE, PRESS R
     ETURN"
1166   GET R$
1170   GOTO 1160
1200   REM ====================
1201   REM =  EXIT THE EDITOR  =
1202   REM ====================
1205   IF FI$ = "" THEN 1250
1210   PRINT D$
1215   PRINT D$;"CLOSE ";FI$
1220   PRINT D$;"CLOSE ";FO$
1250   TEXT : HOME : END
1300   REM ======================
     =
1301   REM =      HELP ROUTINE
     =
1302   REM ======================
     =
1310   PRINT "WHICH COMMAND? ";: GET
     CM$: PRINT CM$
1315   IF  ASC (CM$) =  ASC ("?") THEN
       GOSUB 3500: GOTO 200
1320 TY =  ASC (CM$) -  ASC ("@")

1325   IF TY <  = 0 OR TY > 26 THEN
     250
1330   ON TY GOSUB 1350,1350,2100,
     2150,2200,1350,1350,2350,240
     0,1350,1350,2550,1350,1350,2
     700,2750,1350,2850,2900,2950
     ,1350,1350,1350,1350,1350,13
     50
```

```
1348  PRINT
1349  GOTO 200
1350  PRINT G$;"      UNKNOWN COMM
     AND"
1399  RETURN
1400  REM ==================
1401  REM =  INSERT A LINE  =
1402  REM ==================
1405  PRINT "ENTERING INSERT MODE
     ..."
1410  PRINT "TO QUIT, ENTER '$' A
     S INPUTLINE"
1415 T = 1
1420  INPUT LI$
1425  IF LI$ <  > "$" THEN 1440
1430 T = 0:LI$ = "": GOTO 200
1440  GOSUB 1600
1445  GOTO 1420
1500  REM ====================
1501  REM = OPEN INPUT FILE  =
1502  REM ====================
1505  PRINT "OPEN INPUT FILE"
1510  INPUT "INPUT FILE NAME===>"
     ;FI$
1515  PRINT D$;"OPEN ";FI$
1520  PRINT D$
1525  PRINT "       OPEN OUTPUT F
     ILE"
1530  INPUT "OUTPUT FILE NAME===>
     ";FO$
1535  IF FO$ = FI$ THEN 1530
1540  PRINT D$;"OPEN ";FO$
1545  PRINT D$
1549  GOTO 200
1600  REM ======================
1601  REM = PUT LINE IN OUTPUT =
1602  REM ======================
1605  PRINT D$
1610  PRINT D$;"WRITE ";FO$
1615  IF LI$ <  > "" THEN  PRINT
     LI$
1620  PRINT D$
1645  IF T = 1 THEN  RETURN
1649  GOTO 200
1700  REM ======================
     ===
1701  REM = READ AND DISPLAY INPU
     T =
1702  REM ======================
     ===
1705  PRINT D$
1710  PRINT D$;"READ ";FI$
1715  INPUT LI$
1720  PRINT D$
1725  PRINT LI$
1745  IF T = 1 THEN  RETURN
1749  GOTO 200
1800  REM ==================
1801  REM = TRANSFER LINES =
1802  REM ==================
1805  PRINT "ENTERING TRANSFER MO
     DE..."
1810  PRINT "TO CANCEL, HIT ESCAP
     E"
1812 T = 1
1815  GOSUB 1700
1820  PRINT "TO TRANSFER, HIT THE
      SPACE BAR"
1825  GET AN$
1830  IF AN$ = " " THEN  GOSUB 16
     00
1835  IF AN$ <  > CHR$ (27) THEN
     1815
1840 T = 0: GOTO 200
1900  REM ======================

1901  REM = SHOW EDITING STATUS =

1902  REM ======================

1905  PRINT : PRINT "CURRENT LINE
      IS..."
1910  PRINT "'";LI$;"'"
1915  PRINT "INPUT FILE IS--->'";
     FI$;"'"
1920  PRINT "OUTPUT FILE IS-->'";
     FO$;"'"
1949  GOTO 200
2100  REM ======================
     =
2101  REM = HELP FOR 'C' COMMAND
     =
2102  REM ======================
     =
2105  PRINT : PRINT "C - CLOSE FI
     LES.": PRINT
2107  PRINT "THIS COMMAND TRANSFE
     RS THE REMAINDER OF"
2109  PRINT "THE INPUT FILE TO TH
     E OUTPUT FILE AND"
2111  PRINT "ASKS FOR CONFIRMATIO
     N OF THE CLOSE."
2113  PRINT "ANY ATTEMPT TO 'R' O
     R 'P' OR 'T' AFTER"
2115  PRINT "A 'C' WILL RESULT IN
      A FATAL ERROR."
2149  RETURN
2150  REM ======================
```

LISTING 8.3 (*cont.*)

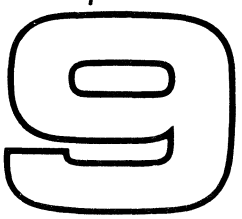
```
      =
2151  REM = HELP FOR 'D' COMMAND
      =
2152  REM ========================
      =
2155  PRINT : PRINT "D - DELETE C
      URRENT LINE.": PRINT
2157  PRINT "THIS COMMAND DELETES
       THE CURRENT LINE."
2159  PRINT "AFTER EXECUTING A 'D
      ' COMMAND, ANY 'P'"
2161  PRINT "COMMAND WILL HAVE NO
       EFFECT, UNTIL A"
2163  PRINT "COMMAND IS ISSUED WH
      ICH CREATES A NEW"
2165  PRINT "CURRENT LINE.  SUCH
      COMMANDS ARE:"
2167  PRINT " 'I', 'T', OR 'R'."
2199  RETURN
2200  REM ========================
      =
2201  REM = HELP FOR 'E' COMMAND
      =
2202  REM ========================
      =
2205  PRINT : PRINT "E - EXIT THE
       EDITOR.": PRINT
2207  PRINT " THIS COMMAND CLEARS
       THE SCREEN AND "
2209  PRINT "RETURNS THE USER TO
      APPLESOFT."
2211  PRINT "ANY OPEN FILES ARE C
      LOSED BEFORE THE"
2213  PRINT "EDITOR QUITS."
2249  RETURN
2350  REM ========================
      =
2351  REM = HELP FOR 'H' COMMAND
      =
2352  REM ========================
      =
2355  PRINT : PRINT "H - ON-LINE
      HELP.": PRINT
2357  PRINT " THIS COMMAND ALLOWS
       THE USER TO GET"
2359  PRINT "A BRIEF EXPLANATION
      OF A GIVEN COMMAND."
2361  PRINT "IT WILL PROMPT BY SA
      YING:"
2363  PRINT "WHICH COMMAND?  A SI
      NGLE LETTER ANSWER"
2365  PRINT "CAUSES INFORMATION O
      N THAT COMMAND TO"
2367  PRINT "BE FORTHCOMING.  THE
       AVAILABLE LETTERS"
2369  PRINT "ARE ALWAYS DISPLAYED
       AT THE TOP OF THE"
2371  PRINT "SCREEN."
2399  RETURN
2400  REM ========================
      =
2401  REM = HELP FOR 'I' COMMAND
      =
2402  REM ========================
      =
2405  PRINT : PRINT "I - INSERT L
      INES.": PRINT
2407  PRINT " THIS COMMAND CAUSES
       THE EDITOR TO GO "
2409  PRINT "INTO 'INSERT MODE'.
       THE USER MAY TYPE"
2411  PRINT "IN AS MANY LINES AS
      DESIRED.  THESE ARE"
2413  PRINT "ADDED TO THE FILE AT
       THE CURRENT EDIT"
2415  PRINT "POSITION.  WHEN FINI
      SHED, THE USER MUST"
2417  PRINT "ENTER '$' AS THE INP
      UT LINE."
2449  RETURN
2550  REM ========================
      =
2551  REM = HELP FOR 'L' COMMAND
      =
2552  REM ========================
      =
2555  PRINT : PRINT "L - LIST THE
       INPUT FILE.": PRINT
2557  PRINT " THIS COMMANDS LISTS
       THE REMAINDER OF"
2559  PRINT "THE INPUT FILE.  THE
       USER MAY GET A"
2561  PRINT "COPY OF THE FILE BY
      FIRST USING THE ?"
2563  PRINT "COMMAND TO TELL DOS
      TO TURN ON THE "
2565  PRINT "PRINTER.  THIS COMMA
      ND WILL RESULT IN"
2567  PRINT "ALL FILES BEING POSI
      TIONED AT THEIR"
2569  PRINT "ENDPOINT.  THE USER
      MAY APPEND LINES TO"
2571  PRINT "AN EXISTING FILE BY
      DOING AN 'O' AND"
2573  PRINT "AN 'L', THEN DECLINI
      NG TO CLOSE THE"
```

```
2575  PRINT "FILES WHEN PROMPTED.
      "
2599  RETURN
2700  REM ======================
      =
2701  REM = HELP FOR 'O' COMMAND
      =
2702  REM ======================
      =
2705  PRINT : PRINT "O - OPEN FIL
      ES.": PRINT
2707  PRINT " THIS COMMAND PROMPT
      S THE USER FOR TWO"
2709  PRINT "FILE NAMES; ONE FOR
      INPUT AND ONE FOR"
2711  PRINT "OUTPUT.  EITHER FILE
       MAY BE NON-EXISTENT";
2713  PRINT "IN WHICH CASE, THEY
      WILL BE CREATED BY"
2715  PRINT "THE EXECUTION OF THE
       OPEN COMMAND."
2749  RETURN
2750  REM ======================
      =
2751  REM = HELP FOR 'P' COMMAND
      =
2752  REM ======================
      =
2755  PRINT : PRINT "P - PUT THE
      CURRENT LINE.": PRINT
2757  PRINT " THIS COMMAND PLACES
       THE CURRENT LINE"
2759  PRINT "INTO THE OUTPUT FILE
      .  IT IS USED IN"
2761  PRINT "MOST CASES FOLLOWING
       A 'R'EAD COMMAND"
2763  PRINT "WHEN THE USER WISHES
       TO ECHO THE LINE"
2765  PRINT "TO THE OUTPUT.  IT I
      S ALSO USEFUL WHEN"
2767  PRINT "IT IS DESIRED TO PLA
      CE SEVERAL COPIES"
2769  PRINT "OF THE SAME LINE INT
      O THE OUTPUT FILE."
2799  RETURN
2850  REM ======================
      =
2851  REM = HELP FOR 'R' COMMAND
      =
2852  REM ======================
      =
2855  PRINT : PRINT "R - READ A L
      INE.": PRINT
2857  PRINT " THIS COMMAND READS
      THE NEXT SEQUENTIAL"
2859  PRINT "LINE FROM THE INPUT
      FILE, DISPLAYS IT"
2861  PRINT "ON THE SCREEN AND MA
      KES IT THE CURRENT"
2863  PRINT "LINE."
2899  RETURN
2900  REM ======================
      =
2901  REM = HELP FOR 'S' COMMAND
      =
2902  REM ======================
      =
2905  PRINT : PRINT "S - SHOW THE
       EDITOR'S STATUS": PRINT
2907  PRINT " THIS COMMAND DISPLA
      YS INFORMATION OF"
2911  PRINT "INTEREST TO THE USER
      .  CURRENTLY THE"
2913  PRINT "INFORMATION DISPLAYE
      D IS:"
2915  PRINT : PRINT "     CURRENT
       LINE"
2917  PRINT "     INPUT FILENAME"

2919  PRINT "     OUTPUT FILENAME
      "
2949  RETURN
2950  REM ======================
      =
2951  REM = HELP FOR 'T' COMMAND
      =
2952  REM ======================
      =
2955  PRINT : PRINT "T - TRANSFER
       LINES.": PRINT
2957  PRINT " THIS COMMAND PUTS T
      HE EDITOR INTO"
2959  PRINT "'TRANSFER' MODE.  LI
      NES ARE READ FROM"
2961  PRINT "INPUT FILE, AND THE
      USER IS GIVEN THE"
2963  PRINT "CHOICE OF EITHER PAS
      SING THEM ON TO THE"
2965  PRINT "OUTPUT FILE, OR SKIP
      PING OVER THEM."
2969  PRINT "THE ESCAPE KEY IS US
      ED TO 'ESCAPE' FROM"
2971  PRINT "TRANSFER MODE."
2999  RETURN
3500  REM ======================
      =
```

LISTING 8.3 (*cont.*)

```
3501  REM = HELP FOR '?' COMMAND
      =
3502  REM ========================
      =
3505  PRINT : PRINT "? - ISSUE DO
      S COMMANDS": PRINT
3507  PRINT " THIS COMMAND ALLOWS
       THE USER TO ISSUE"
3509  PRINT "ANY DIRECT DOS COMMA
      ND FROM THE EDITOR."
3511  PRINT "THIS IS USEFUL FOR V
      IEWING THE CATALOG,"
3513  PRINT "DELETING FILES, OR R
      ENAMING FILES."
3514  PRINT
3515  PRINT "***** ONE WARNING, H
      OWEVER:": PRINT
3517  PRINT "IF YOU WISH TO RENAM
      E A FILE, THEN THE"
3519  PRINT "COMMAND MUST BE ENCL
      OSED IN DOUBLE"
3521  PRINT "QUOTES, OR APPLESOFT
       WILL REFUSE TO "
3523  PRINT "READ PAST THE COMMA
      BETWEEN THE TWO "
3525  PRINT "FILENAMES."
3549  RETURN
30000  REM ======================
      ==
30001  REM = SET SCROLLING WINDOW
       =
30002  REM ======================
      ==
30005  POKE 32,0: POKE 33,40
30010  POKE 34,8: POKE 35,23
30015  HOME : RETURN

]
```

Chapter



String Arrays

1. DECLARATION AND USE OF STRING ARRAYS

APPLESOFT BASIC provides considerably more string manipulation facilities than does Integer BASIC. Whereas Integer BASIC implements each string as a character array and does not have arrays of strings, APPLESOFT BASIC implements individual strings as simple variables and allows arrays of string variables.

An APPLESOFT string array is declared with a DIM statement:

```
10 DIM ST$(100),LI$(50)
```

The program of Listing 9.1 at the end of the chapter presents an example of string manipulation using string arrays. The program sorts an array of strings into alphabetical order, using the Bubble Sort algorithm.

2. LANGUAGE MANIPULATION

Several years ago there was a popular parlor game known as "Mad Libs." In this game, one of several stories is chosen by the players. The story is incomplete, and at several points in the story there are missing words represented by blanks. Each blank is categorized by a part of speech such as "noun," "verb," "adjective," and so on. The players supply words, usually as outrageous as possible, to fill in the blanks, without knowing the context into which the words will fit. The story is then read aloud with the suggested words used to fill in the blanks. The results are unexpected and it is hoped, hilarious.

In this section we present a program that "plays" a somewhat similar game. It takes phrases: subject phrases, gerunds, nouns, infinitive verbs, etc., and uses them to build random analogies. It generates such wacky analogies as: "Writing a book about computers is like singing to a bellybutton."

The program uses a vocabulary of predefined words and phrases to generate the analogies. The sentence structure of the analogies is defined by the order in which the various words and phrases are chosen. It is possible to obtain different styles of analogies by changing this ordering. Figure 9.1 describes the overall "shape" of an analogy.

The interpretation of the diagrams in Figure 9.1 is as follows. An analogy is made up by stringing together several individual elements. These elements are indicated

in the diagram by sequentially following the arrows. The elements in ovals are variable; the elements in the rectangles are fixed. Several arrows emanating from a small circle (o) represent a choice. Thus, an analogy is formed by:

A subject phrase, followed by the words
"is like," followed by
a gerund, followed by
a noun phrase, followed by
zero or more qualifying phrases.

The possible repetition of the qualifying phrase is indicated in the diagram by the arrow flowing out of the corresponding oval and flowing back to a point before the choice circle.

A qualifying phrase itself possesses some structure as is indicated by the second part of Figure 9.1. It may be either:

An infinitive verb followed by a noun phrase;
A temporal connective followed by a gerund, followed by a noun phrase;
A spatial connective followed by a noun phrase.

Here are some examples of the kinds of words and phrases which might fit each of the various categories of vocabulary:

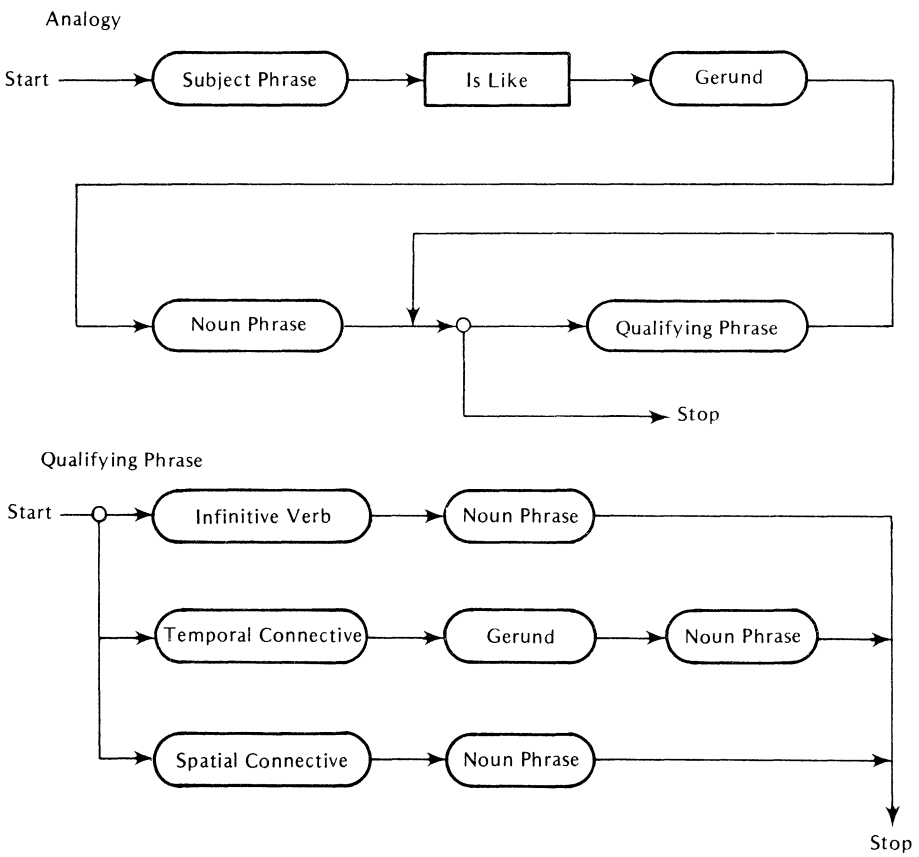
Subject Phrases

Riding a bicycle in a tornado
Writing a book about computers
Losing your job
Teasing a rhinoceros
Finding a needle in a haystack
Eating raw fish
Having 17 children
Forgetting your wife's birthday
Falling into a mud puddle

Gerunds

working for

FIGURE 9.1 Structure of an Analogy



reading to
singing to
losing
kicking
swatting
investigating
running into

Noun Phrases

a bellybutton
a hot-air balloon
the New York Philharmonic
a tank of laughing gas
a pair of striped pajamas
a brass doorknob
a tricycle
a jug of moonshine
a calculus book
the ring around the bathtub
leftover pizza

Infinitive Verbs

dream about
oil
smell
slug
toast
bury
squeeze
paint
decorate

Temporal Connectives

before
while
in preparation for
because of
instead of
after

Spatial Connectives

on top of
hand in hand with
underneath
beside
together with
cheek to cheek with

A lot of the fun of using the analogies generator lies in making up the vocabulary. You can spend hours coming up with new phrases and clever twists, then sitting back and watching the unexpected combinations which they lead to.

Two versions of the analogies generator are presented in Listings 9.2 and 9.3. The first version uses external files to store the vocabulary. There is a separate file for each type of element—the file name being identical to the element name. For example, there are files called SUBJECT PHRASES, GERUNDS, and INFINITIVES. The program manages to read all of these files and to store their contents in different arrays by the use of the concept of a buffer array.

Reading many files with one subroutine

The subroutine beginning at line 1000 reads a file of strings into a string array called X$. This array is not the final resting place of the strings being read, but merely a waystation or buffer. Its use allows the same subroutine to be called in order to read each of the separate files required by the analogies program. Once the contents of a given file have been read into X$, they are copied into the correct final array by the caller of the file read subroutine. For example, in lines 110 to 115 we see:

```
110 F$ = "GERUNDS": GOSUB 1000 : L2 =
    CO - 1
115 FOR I = 0 TO L2: GE$(I) = X$(I) : NEXT I
```

which causes the file GERUNDS to be read and its strings to be stored in the array GE$.

The second version of the program uses DATA statements to store the vocabulary. It READs in the vocabulary in the subroutine located at lines 100 to 199. It utilizes a trick in order to differentiate the different categories of vocabulary items being read.

Using "control strings" in input data

In order for the vocabulary items to be read from the DATA statements of the program into the appropriate arrays, some way must be provided to differentiate the various categories from one another.

One way of doing this would simply be to require an equal and fixed number of items in each category and to arrange them in some predetermined order in the DATA statements. This has the disadvantage of inflexibility—

once the original DATA statements have been written into the program, additions to any of the categories of vocabulary items would require that everything be reentered: very tough on the two-fingered keyboard artists of the world.

Another, more flexible technique is used in the program of Listing 9.3. Extra items are placed in the DATA statements which serve to tell the READing subroutine which kind of vocabulary items will follow in a particular section of DATA statements. The extra items are one of the following strings, each of which signals the beginning of a section of vocabulary items of a particular type:

$S	SUBJECT PHRASES
$G	GERUNDS
$N	NOUNS
$T	TEMPORAL CONNECTIVES
$P	SPATIAL CONNECTIVES
$I	INFINITIVES

Thus, for example, the DATA statement in line 9013 begins with the string $I. This tells the program that the following strings will be infinitives. The effect of a control string lasts until another one is encountered later on.

To add new vocabulary items using this method is easy. You simply write some more DATA statements, making sure to include the appropriate control strings.

Using the Analogies Program

You can use the Editor program of Chapter 8 to create files with your own vocabularies, or you can modify the DATA version of the program to add your own words and phrases. Here's your chance to use your imagination, wit, and charm!

3. RANDOM NUMBER GENERATION

The Analogies program uses the RND function to select randomly from among the phrases in its vocabulary. The variety of the resulting analogies depends in part on the quality of the underlying random number generator. It is beyond the scope of this book to discuss precise mathematical techniques for evaluating random number generators, but we shall present an informal technique in this section which is appealing to common sense. It does require a little recall of high-school algebra, however, so be forewarned!

Figure 9.2 shows a circle inside a square, drawn using

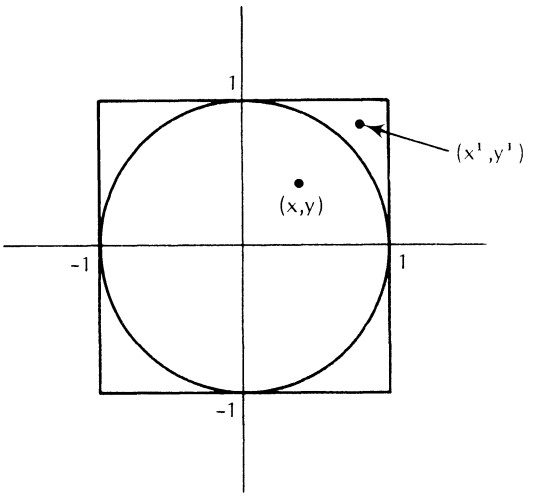


FIGURE 9.2 Approximation of Using Pairs of Random Numbers

rectangular coordinates. The circle has a radius of 1. Consequently, its area is equal to $\pi(1)^2 = \pi$. The square has sides of length 2. Therefore its area is equal to $2 \times 2 = 4$. The ratio of the area of the circle to the area of the square is $\pi/4$. Since the figure is divided symmetrically into four congruent parts, the ratio of the area of the circle to the area of the square within any one of the four parts (or "quadrants" as the mathematicians call them) is also $\pi/4$.

Now think about the RND function. It generates numbers between 0 and 1.

0 <= RND(N) < 1.0

If we use RND to generate a pair of numbers:

X = RND(N)
Y = RND(N)

and then treat the pair (X,Y) as the coordinates of a point on the graph of Figure 9.2, that point will lie *inside* the square. If the numbers X and Y are a truly random pair and we generate a large number of such pairs, then the proportion of pairs which represent points inside the circle to all pairs should approximate the ratio of the area of the circle to the area of the square, or $\pi/4$.

What is the criterion for a pair (X,Y) to represent a point which lies inside the circle? Recall from algebra that the equation for the circumference of this circle itself is:

$$X^2 + Y^2 = 1$$

The points which lie *inside* the circle will satisfy the condition:

$$X^2 + Y^2 < 1$$

This is a condition which we can easily translate into BASIC. The program of Listing 9.4 is an attempt to

approximate the value of π using the ideas of the previous discussion. Several pairs (X,Y) of numbers are generated using the APPLESOFT RND function. Each of these pairs is counted as being inside the square, since both X and Y are less than 1. If X*X + Y*Y < 1.0, then the pair is counted as being inside the circle. The ratio of those inside the circle to all those generated approximates $\pi/4$. So four times that ratio approximates π.

It is interesting—to some people at least—to sit and watch the series of numbers being printed out to see how close an approximation is reached. The closer the approximation, the "better" in some sense of word, is the quality of the random number generator. It should also be interesting to try the same program on a number of different systems in order to compare the results.

The program of Listing 9.4 was run several times with the outcomes shown in Table 9.1. We shall allow you to draw your own conclusions from this data, and from your experiences using the analogies program.

TABLE 9.1

N	H	π
100	84	3.36
100	77	3.08
100	80	3.2
100	77	3.08
100	80	3.2
1,000	784	3.136
1,000	779	3.116
1,000	776	3.104
1,000	782	3.128
10,000	7723	3.0892
10,000	7722	3.0888
10,000	7723	3.0892
10,000	7722	3.0888
10,000	7723	3.0892

4. USING THE STRING FUNCTIONS

APPLESOFT provides some simple functions for taking strings *apart:*

LEFT$
MID$
RIGHT$

In this section, we present a subroutine which uses some of these functions to break apart a line of text into "words." A word is defined as a consecutive sequence of nonblank characters. A word may occur in one of three ways:

- A sequence of nonblank characters at the beginning of the line.
- A sequence of nonblank characters at the end of the line.
- A sequence of nonblank characters *surrounded* by two blanks.

The subroutine is obliged to detect all three possibilities.

The following variables will be used:

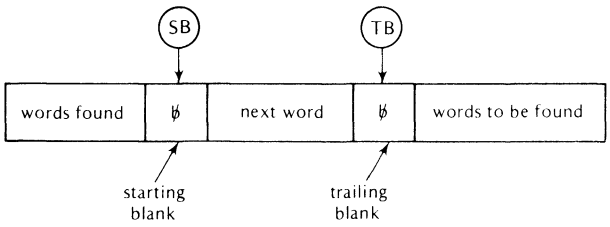
LI$	Original line of text
IL	Length of LI$
WO$	String array to hold the words extracted by the subroutine
NW	The number of words found
SB	Index to find the starting blank of a word
CP	Character pointer; to keep track of how much of LI$ has been examined
C$	The "next" character of LI$; used to detect the trailing blank

We actually present two versions of the subroutine. The first version, shown as Listing 9.5, was written in a relatively ad-hoc fashion. It handles special cases piecemeal and uses two flags not needed by the second version, shown in Listing 9.6.

The second version uses a simplifying technique in order to get rid of the two flag variables and several unnecessary tests. It is based on the use of an *invariant relation* picture. This is a description of the state of the processing at any given point during the course of execution of the subroutine. The idea is to write the code so that the invariant relation picture is always maintained, but progress is made toward the program's ultimate goal. Figure 9.3 shows the invariant relation for the subroutine.

The picture shows that in general there may be words at the beginning of the line which have already been found by the subroutine. Following these words will be the "next" word to be found or perhaps just found. It will be terminated by a trailing blank, and perhaps preceded by a starting blank. The positions of these two blanks in the line are given by the variables SB and CP. Finally, at the right-hand end of the line there will be more words still to be found in the future. The subroutine should guarantee that the "words to be found" part of the picture is eventually made empty.

FIGURE 9.3 Invariant Relation for "Words" Subroutine



There are some problems with this picture, however. It does not handle all cases uniformly. The first and last words in the line are special since they lack a starting and trailing blank, respectively. The starting blank is not essential. It serves only in the calculation of the length of the word found. If we simply start out with SB = 0, the subroutine will behave as if there was an "invisible blank" to the left of the line. The trailing blank is more difficult to deal with. It *is* used by the scan in order to know when the *end* of the next word has been reached. If the last word in the string has no trailing blank, the subroutine could easily try to read beyond the end of the line and cause program errors. To avoid this possibiilty, the first version of the subroutine checks for the *end of the string* by the test:

CP > LEN(LI$)

and relies on this to terminate the scan for the last word.

It would be much simpler if one could *assume* that the last word was always followed by a trailing blank. One way to ensure this would be to *require* that every caller of the subroutine provide a line which ended in one or more blanks. If there are many callers of the subroutine, this could become awkward. It is easier to have the subroutine itself insert the trailing blank, as "protection." In the second version of the subroutine, we see this happening:

115 LI$ = LI$ + " "

This innocuous little statement has surprising ramifications:

1. It is no longer necessary to test explicitly for an empty line. (IF LI$=" " THEN RETURN)
2. The two flag variables TB and ES are not needed any longer.

Since the line is guaranteed to have *at least one* blank, the BASIC statements in the subroutine may assume that:

- The line is nonempty.
- There is at least one trailing blank (although it may follow an empty word).

These assumptions allow simplification. Compare Listings 9.4 and 9.5 to see how many simplifications you can locate.

5. EXPLORATIONS

- Write some other "language generation" programs similar to the analogies generator program. Some possibilities:

  —A beatitudes generator: "Blessed are the —— for they shall ——"

  — A rock group name generator.

  —A monster or science-fiction movie title generator.

  —A help-wanted ad generator.

- Try out variations on the syntax of the analogies generated. See if you can find a way to encode the desired structure of the analogies. This structure would be read in and interpreted by the program to dictate the resulting strings. Can you use your ideas on the previous exploration?
- Write a subroutine to examine a string to find occurrences of numbers. The subroutine could isolate the number in string form and convert it into numeric form in another variable.
- Both versions of the Analogies program presented in this chapter use separate arrays to hold the individual vocabularies. Notice that this necessitates a separate subroutine for randomly selecting each type of vocabulary item. Investigate the savings in program length which results from using a two-dimensional string array to store the vocabulary:

  VO$(100,10)

What application might this have to earlier explorations?

LISTINGS

LISTING 9.1 SORTING STRINGS IN APPLESOFT

```
]LIST

10  DIM A$(100)
11 D$ =  CHR$ (4): REM  CONTROL-D

15  INPUT "INPUT FILE===>";F$
20  REM ========================
     =====
21  REM = READ IN THE DATA AND PR
     INT =
22  REM = THE ORIGINAL ORDER.
        =
23  REM ========================
     =====
25  GOSUB 100
26 AL = I
30  FOR I = 1 TO AL
35  PRINT "A$(";I;")= ";A$(I)
40  NEXT I
50  REM ========================
     ====
51  REM =  BUBBLE SORT THE STRING
     S  =
52  REM ========================
     ====
54  INVERSE : PRINT "SORTING": NORMAL

55  FOR L = 1 TO AL - 1
60  FOR I = 1 TO AL - L
65  IF A$(I) > A$(I + 1) THEN T$ =
     A$(I):A$(I) = A$(I + 1):A$(I
     + 1) = T$
70  NEXT I
75  NEXT L
80  REM ========================

81  REM = PRINT OUT THE RESULTS =

82  REM ========================

85  FOR I = 1 TO AL
90  PRINT "A$(";I;")= ";A$(I)
95  NEXT I
99  END
100  REM ======================
101  REM = READ IN STRING DATA =
102  REM ======================
105  ONERR  GOTO 199
110 I = 1
120  PRINT D$;"OPEN ";F$
125  PRINT D$;"READ ";F$
130  INPUT A$(I)
135  IF A$(I) = "END" THEN  PRINT
     D$;"CLOSE ";F$: RETURN
140 I = I + 1
145  GOTO 130
199 I = I - 1: GOTO 26
```

LISTING 9.2 ANALOGIES GENERATOR

```
]LIST

1  REM ========================
     ==
2  REM =
     =
3  REM =   ANALOGIES GENERATOR
     =
4  REM =              BY
     =
5  REM = DR. RICHARD C. VILE, JR.
     =
6  REM =  ALL COMMERCIAL RIGHTS
     =
7  REM =         RESERVED
     =
8  REM =
     =
9  REM ========================
     ==
10  DIM SU$(100),GE$(100),NO$(100
     ),TC$(50),SC$(50),IN$(100)
11  DIM X$(100)
12 D$ =  CHR$ (4):RN% = 5
13 B$ = ""
14 DE = 1
15  HOME : VTAB 5: HTAB 5: FLASH
     : PRINT "LOADING THE VOCABUL
     ARY": NORMAL
20  PRINT : PRINT : GOSUB 100
25  HOME : VTAB 5: HTAB 5: INPUT
     "HOW MANY ANALOGIES ";NUM
26  PRINT "      ";: INPUT "SCREEN
     OR PRINTER?";WH$
27  IF WH$ = "SCREEN" THEN SC = 1

28  IF WH$ <  > "SCREEN" THEN SC =
     0
29  IF SC = 1 THEN LL = 40
30  IF SC = 0 THEN LL = 72
31  FOR I = 1 TO NUM
35 N =  PEEK (78) + 256.0 *  PEEK
```

LISTING 9.2 (*cont.*)

```
     (79)
36   PRINT N: REM  DEBUG
40   IF NUM < 0 THEN  HOME : END
42   FOR I = 1 TO NUM
44   GOSUB 200
46   NEXT I
50   GOTO 25
99   END
100  F$ = "SUBJECT PHRASES": GOSUB
      1000:L1 = CO - 1
105   FOR I = 0 TO CO:SU$(I) = X$(
     I): NEXT I
110  F$ = "GERUNDS": GOSUB 1000:L2
       = CO - 1
115   FOR I = 0 TO CO:GE$(I) = X$(
     I): NEXT I
120  F$ = "NOUNS": GOSUB 1000:L3 =
     CO - 1
125   FOR I = 0 TO L3:NO$(I) = X$(
     I): NEXT I
130  F$ = "TEMPORAL CONNECTIVES": GOSUB
     1000:L4 = CO - 1
135   FOR I = 0 TO L4:TC$(I) = X$(
     I): NEXT I
140  F$ = "SPATIAL CONNECTIVES": GOSUB
     1000:L5 = CO - 1
145   FOR I = 0 TO L5:SC$(I) = X$(
     I): NEXT I
150  F$ = "INFINITIVES": GOSUB 100
     0:L6 = CO - 1
155   FOR I = 0 TO CO:IN$(I) = X$(
     I): NEXT I
199   RETURN
200   REM ======================
201   REM = GENERATE AN ANALOGY =
202   REM ======================
205  DU = 0: REM  DON'T DUMP EMPTY
      LINE
208   GOSUB 2000
210   GOSUB 4000
215  PH$ = "IS LIKE ": GOSUB 4000
220   GOSUB 2100
225   GOSUB 4000
230   GOSUB 2200
235   GOSUB 4000
240  RG% = 2: GOSUB 3000
245   FOR J = 1 TO RN%
250   GOSUB 2400
255   GOSUB 4000
260   NEXT J
265  DU = 1: GOSUB 4000
298   GOSUB 5100: REM  WAIT
299   RETURN
1000  REM ======================
     =
1001  REM = READ AN INPUT FILE.
     =
1002  REM = MUST END WITH A LINE
     =
1003  REM = CONTAINING "END".
     =
1005  REM ======================
     =
1008  PRINT "     READING ";F$
1010  PRINT D$
1015  PRINT D$;"OPEN ";F$
1020  PRINT D$;"READ ";F$
1025 CO = 0
1030  INPUT X$(CO)
1035 CO = CO + 1
1040  IF X$(CO - 1) = "END" THEN
     1090
1045  GOTO 1030
1090  PRINT D$
1095  PRINT D$;"CLOSE ";F$
1099  RETURN
2000  REM ======================
     ======
2001  REM = GENERATE A SUBJECT PH
     RASE =
2002  REM ======================
     ======
2005 RG% = L1
2010  GOSUB 3000
2015 PH$ = SU$(RN%) + " "
2049  RETURN
2100  REM ====================
2101  REM = GENERATE A GERUND =
2102  REM ====================
2105 RG% = L2
2110  GOSUB 3000
2115 PH$ = GE$(RN%) + " "
2149  RETURN
2200  REM ======================
     ===
2201  REM = GENERATE A NOUN PHRAS
     E =
2202  REM ======================
     ===
2205 RG% = L3
2210  GOSUB 3000
2215 PH$ = NO$(RN%) + " "
2249  RETURN
2300  REM ======================
     ==========
2301  REM = GENERATE AN INFINITIV
     E PHRASE =
2302  REM ======================
```

```
    ==========
2305 RG% = L6
2309 N = 91.7013117
2310  GOSUB 3000
2315 PH$ = "TO " + IN$(RN%) + " "

2319  RETURN
2400  REM ======================
    =========
2401  REM = GENERATE A QUALIFYING
      PHRASE =
2402  REM ======================
    =========
2405 RG% = 3
2410  GOSUB 3000
2415  ON RN% + 1 GOTO 2500,2600,2
    800
2500  GOSUB 2300
2505 TP$ = PH$
2510  GOSUB 2200
2515 PH$ = TP$ + PH$
2549  RETURN
2600  GOSUB 2700
2605 TP$ = PH$: REM  SAVE TEMPORA
    L CONNECTIVE
2610  GOSUB 2100
2615 PH$ = TP$ + PH$
2620 TP$ = PH$
2625  GOSUB 2200
2630 PH$ = TP$ + PH$
2649  RETURN
2700  REM ==================
2701  REM = PICK A TEMPORAL =
2702  REM =    CONNECTIVE    =
2703  REM ==================
2705 RG% = L4: GOSUB 3000
2710 PH$ = TC$(RN%) + " "
2749  RETURN
2800  GOSUB 2900
2805 TP$ = PH$: REM  SAVE SPATIAL
      CONNECTIVE
2810  GOSUB 2200
2815 PH$ = TP$ + PH$
2849  RETURN
2900  REM ==================
2901  REM = PICK A SPATIAL =
2902  REM =   CONNECTIVE    =
2903  REM ==================
2910 RG% = L5
2915  GOSUB 3000
2920 PH$ = SC$(RN%) + " "
2949  RETURN
3000  REM ======================
    =====
3001  REM = GENERATE A RANDOM NUM
    BER =
3002  REM = BETWEEN 0 AND RG%-1.
        =
3003  REM ======================
    =====
3005 RN% =  INT ( RND (N) * RG%)
3049  RETURN
4000  REM ======================
    =
4001  REM = ACCUMULATE AND PRINT
    =
4002  REM = THE ANALOGY.
    =
4003  REM ======================
    =
4005  IF  LEN (B$) +  LEN (PH$) >
    LL THEN  GOSUB 4100
4010 B$ = B$ + PH$
4015  IF DU = 1 THEN B$ = "": GOSUB
    4100
4049  RETURN
4100  REM ======================
    =
4101  REM = DIRECT OUTPUT STREAM
    =
4102  REM ======================
    =
4110  IF SC = 1 THEN  GOSUB 4300
4115  IF SC = 0 THEN  GOSUB 4200
4149  RETURN
4200  REM ======================

4201  REM = DUMP TO THE PRINTER =

4202  REM ======================

4205  PRINT D$
4210  PRINT D$;"PR#1"
4215  PRINT  CHR$ (9);"72N";
4220  PRINT B$;
4222  IF DU = 1 THEN  PRINT
4225  PRINT D$
4230  PRINT D$;"PR#0"
4235 B$ = ""
4249  RETURN
4300  REM ======================
4301  REM = DUMP TO THE SCREEN =
4302  REM ======================
4310  PRINT B$
4315  IF DU = 1 THEN  PRINT
4320 B$ = ""
4349  RETURN
5000  REM ======================
```

LISTING 9.2 *(cont.)*

```
5001  REM = DEBUG PRINTOUT OF =
5002  REM = VALUE OF RN%.     =
5003  REM ====================
5004  RETURN
5005  IF DE = 1 THEN  PRINT RN%;"
      ";
5099  RETURN
5100  REM ========================
      ==
5101  REM = PAUSE TO READ ANALOGY
      =
5102  REM ========================
      ==
5110  IF SC = 0 THEN  RETURN
5115  PRINT : PRINT "TO CONTINUE,
      PRESS RETURN"
5120  INPUT R$
5149  RETURN
```

LISTING 9.3 ANALOGIES WITH DATA STATEMENTS

```
]LIST

1  REM ============================
     ==
2  REM =
     =
3  REM =    ANALOGIES WITH DATA
     =
4  REM =              BY
     =
5  REM = DR. RICHARD C. VILE, JR.
     =
6  REM =  ALL COMMERCIAL RIGHTS
     =
7  REM =        RESERVED
     =
8  REM =
     =
9  REM ============================
     ==
10  DIM SU$(100),GE$(100),N1$(100
    ),CN$(100),IN$(100)
11  DIM X$(100)
12 D$ =  CHR$ (4):RN% = 5
13 B$ = ""
14 DE = 1
15  HOME : VTAB 5: HTAB 5: FLASH
    : PRINT "LOADING THE VOCABUL
    ARY": NORMAL
18 L1 = 1:L2 = 1:L3 = 1:L4 = 1:L5
     = 1:TY = 1
20  PRINT : PRINT : GOSUB 100
25  HOME : VTAB 5: HTAB 5: INPUT
    "HOW MANY ANALOGIES ";NUM
26  IF NUM <  = 0 THEN  HOME : END

30  PRINT "    ";: INPUT "SCREEN
    OR PRINTER(S/P)?";WH$
32  IF WH$ = "S" THEN SC = 1
34  IF WH$ = "SCREEN" THEN SC = 1

36  IF WH$ <  > "S" AND WH$ <  >
    "SCREEN" THEN SC = 0
45  IF SC = 1 THEN LL = 40
47  IF SC = 0 THEN LL = 72
50  FOR I = 1 TO NUM
55 N =  PEEK (78) + 256.0 *  PEEK
    (79)
56  PRINT N: REM  DEBUG
60  GOSUB 200
65  NEXT I
70  GOTO 25
99  END
100  REM ========================
     =====
101  REM = READ VOCABULARY FROM D
     ATA =
102  REM ========================
     =====
105  READ L$: REM  GET A STRING
106  IF L$ = "END" THEN 190
108  PRINT L$
110  IF  LEFT$ (L$,1) = "$" THEN
      GOSUB 8000: GOTO 105
112  ON TY GOTO 120,130,140,150,1
     60
115  GOTO 105
120 SU$(L1) = L$
122 L1 = L1 + 1
125  GOTO 105
130 GE$(L2) = L$
132 L2 = L2 + 1
135  GOTO 105
140 N1$(L3) = L$
142 L3 = L3 + 1
145  GOTO 105
150 CN$(L4) = L$
152 L4 = L4 + 1
155  GOTO 105
160 IN$(L5) = L$
162 L5 = L5 + 1
165  GOTO 105
190 L1 = L1 - 1:L2 = L2 - 1
192 L3 = L3 - 1:L4 = L4 - 1:L5 =
```

```
    L5 - 1
199  RETURN
200  REM ========================
201  REM = GENERATE AN ANALOGY =
202  REM ========================
205 DU = 0: REM  DON'T DUMP EMPTY
     LINE
208  GOSUB 2000
210  GOSUB 4000
215 PH$ = "IS LIKE ": GOSUB 4000
220  GOSUB 2100
225  GOSUB 4000
230  GOSUB 2200
235  GOSUB 4000
240 RG% = 2: GOSUB 3000
245  FOR J = 1 TO RN%
250  GOSUB 2400
255  GOSUB 4000
260  NEXT J
265 DU = 1: GOSUB 4000
298  GOSUB 5100: REM  WAIT
299  RETURN
2000  REM ========================
     ======
2001  REM = GENERATE A SUBJECT PH
     RASE =
2002  REM ========================
     ======
2005 RG% = L1
2010  GOSUB 3000
2015 PH$ = SU$(RN%) + " "
2049  RETURN
2100  REM =====================
2101  REM = GENERATE A GERUND =
2102  REM =====================
2105 RG% = L2
2110  GOSUB 3000
2115 PH$ = GE$(RN%) + " "
2149  RETURN
2200  REM ========================
     ===
2201  REM = GENERATE A NOUN PHRAS
     E =
2202  REM ========================
     ===
2205 RG% = L3
2210  GOSUB 3000
2215 PH$ = N1$(RN%) + " "
2299  RETURN
2300  REM ========================
     ==========
2301  REM = GENERATE AN INFINITIV
     E PHRASE =
2302  REM ========================
     ==========
2305 RG% = L5
2309 N = 91.7013117
2310  GOSUB 3000
2315 PH$ = "TO " + IN$(RN%) + " "

2319  RETURN
2400  REM ========================
     =========
2401  REM = GENERATE A QUALIFYING
     PHRASE =
2402  REM ========================
     =========
2405 RG% = 2
2410  GOSUB 3000
2415  ON RN% + 1 GOTO 2500,2600
2500  GOSUB 2300
2505 TP$ = PH$
2510  GOSUB 2200
2515 PH$ = TP$ + PH$
2549  RETURN
2600  GOSUB 2700
2605 TP$ = PH$: REM  SAVE CONNECT
     IVE
2610  GOSUB 2100
2615 PH$ = TP$ + PH$
2620 TP$ = PH$
2625  GOSUB 2200
2630 PH$ = TP$ + PH$
2649  RETURN
2700 RG% = L4
2705  GOSUB 3000
2710 PH$ = CN$(RN%) + " "
2749  RETURN
3000  REM ========================
     =====
3001  REM = GENERATE A RANDOM NUM
     BER =
3002  REM = BETWEEN 0 AND RG%-1.
        =
3003  REM ========================
     =====
3005 RN% =  INT ( RND (N) * RG%)
3010  IF RN% = 0 THEN 3005
3049  RETURN
4000  REM ========================
     =
4001  REM = ACCUMULATE AND PRINT
     =
4002  REM = THE ANALOGY.
     =
4003  REM ========================
     =
4005  IF  LEN (B$) +  LEN (PH$) >
```

LISTING 9.3 (cont.)

```
     LL THEN  GOSUB 4100
4010 B$ = B$ + PH$
4015  IF DU = 1 THEN B$ = "": GOSUB
     4100
4049  RETURN
4100  REM ======================
     =
4101  REM = DIRECT OUTPUT STREAM
     =
4102  REM ======================
     =
4110  IF SC = 1 THEN  GOSUB 4300
4115  IF SC = 0 THEN  GOSUB 4200
4149  RETURN
4200  REM ======================

4201  REM = DUMP TO THE PRINTER =

4202  REM ======================

4205  PRINT D$
4210  PRINT D$;"PR#1"
4215  PRINT  CHR$ (9);"72N";
4220  PRINT B$;
4222  IF DU = 1 THEN  PRINT
4225  PRINT D$
4230  PRINT D$;"PR#0"
4235 B$ = ""
4249  RETURN
4300  REM ======================
4301  REM = DUMP TO THE SCREEN =
4302  REM ======================
4310  PRINT B$
4315  IF DU = 1 THEN  PRINT
4320 B$ = ""
4349  RETURN
5000  REM ======================
5001  REM = DEBUG PRINTOUT OF =
5002  REM = VALUE OF RN%.     =
5003  REM ======================
5005  IF DE = 1 THEN  PRINT RN%;"
     ";
5099  RETURN
5100  REM ======================
     ==
5101  REM = PAUSE TO READ ANALOGY
     =
5102  REM ======================
     ==
5110  IF SC = 0 THEN  RETURN
5115  PRINT : PRINT "TO CONTINUE,
     PRESS RETURN"
5120  INPUT R$
5149  RETURN
8000  REM ======================
     ==
8001  REM = CATEGORIZE INPUT TYPE
     =
8002  REM ======================
     ==
8010 TY = 1
8012 T$ =  MID$ (L$,2,1)
8015  IF T$ = "S" THEN TY = 1
8016  IF T$ = "G" THEN TY = 2
8017  IF T$ = "N" THEN TY = 3
8018  IF T$ = "C" THEN TY = 4
8019  IF T$ = "I" THEN TY = 5
8049  RETURN
9000  REM ======================
     ====
9001  REM = DATA STATEMENTS FOR T
     HE =
9002  REM = ANALOGIES VOCABULARY.
        =
9003  REM ======================
     ====
9005  DATA  $S,WINNING THE LOTTER
     Y
9006  DATA  $S,SWIMMING IN HOT GL
     UE
9007  DATA  $S,HANG GLIDING ON MT
     . ST. HELENS
9008  DATA  $G,SWALLOWING
9009  DATA  $G,RUNNING AFTER
9010  DATA  $G,LISTENING TO
9011  DATA  $S,WRITING A BOOK ABO
     UT COMPUTERS
9012  DATA  $G,MISUNDERSTANDING
9013  DATA  $I,JUGGLE,BEND,INFECT
     ,SQUEEZE,TICKLE,STRAIN,SWIPE
     ,NESTLE,UNDERMINE
9014  DATA  $S,TAKING YOUR PET EL
     EPHANT FOR A 'WALK'
9015  DATA  $N,A RUBBER CHICKEN,N
     IKITA KHRUSCHEV, A BERMUDA O
     NION, LAUGHING GAS,AN OLD UN
     DERSHIRT
9016  DATA  $C,INSTEAD OF, TO AVO
     ID, WHILE, DURING, IN PREPAR
     ATION FOR, BECAUSE OF,BEFORE
     , AFTER
9017  DATA  $I,SQUEEZE,RATTLE, SH
     AKE,INSPECT,INFURIATE,SWALLO
     W
9018  DATA $G,DINING OUT WITH,PAI
     NTING
9019  DATA  $G,TELEPHONING,NEGOTI
```

```
     ATING WITH,JUMPING OVER
9020  DATA  $N,THE PENTAGON, A RE
     D TOUPEE, A GALLON OF PICKLE
     S, A TICKET TO THE WORLD SER
     IES
9099  DATA  END

]&F
```

LISTING 9.4 RND FUNCTION TEST

```
]LIST

1  REM ================================
2  REM =                              =
3  REM =        RND  TESTER           =
4  REM =           BY                 =
5  REM = DR. RICHARD C. VILE, JR.     =
6  REM =                              =
7  REM ================================
10  INPUT "STOP? ";ST
11 N = 0:H = 0
12 K =  PEEK (78) + 256.0 *  PEEK (79)
15 X =  RND (K):Y =  RND (K)
16 N = N + 1
20  IF X * X + Y * Y < 1.0 THEN H = H + 1
25  PRINT "N= ";N;"    H= ";H;"    PI= ";4 * H / N
30  IF N < ST THEN 15
40  PRINT  CHR$ (7);
41  GOTO 40
99  END

]
```

LISTING 9.5 TOKENIZE

```
]LIST

10  DIM WO$(100)
20  PRINT "GIMME A STRING"
25  INPUT LI$
28  GOSUB 100
30  FOR I = 1 TO NW
35  PRINT WO$(I)
40  NEXT I
55  GOTO 20
100  REM ========================
     ===
101  REM = EXTRACT WORDS FROM LIN
     E =
102  REM =
       =
103  REM =  INPUT: LI$ - LINE OF
       =
104  REM =         TEXT.
       =
105  REM = OUTPUT: NW - NUMBER OF
       =
106  REM =         WORDS FOUND.
       =
107  REM =         WO$ - WORDS
       =
108  REM =         FOUND STORED I
     N =
109  REM =         STRING ARRAY.
       =
```

LISTING 9.5 *(cont.)*

```
110  REM =
       =
111  REM ========================
     ===
115  IF LI$ = "" THEN  RETURN : REM
      REJECT EMPTY STRING
119  REM ========================
     =====
120  REM = SET UP POINTERS AND FL
     AGS =
121  REM ========================
     =====
125 SB = 0:CP = 1:ES = 0
126 NW = 0
130 TB = 0: REM  SET UP TO EXTRAC
     T NEXT WORD
134  REM ======================
135  REM = GET NEXT CHARACTER =
136  REM ======================
140 C$ =  MID$ (LI$,CP,1)
144  REM ========================
     ====
145  REM = CHECK FOR TRAILING BLA
     NK =
146  REM ========================
     ====
150  IF C$ = " " THEN TB = 1
154  REM ========================
     ===
155  REM = CHECK FOR END OF STRIN
     G =
156  REM ========================
     ===
160  IF CP >  =  LEN (LI$) THEN E
     S = 1
164 CP = CP + 1
165  IF ( NOT TB) AND ( NOT ES) THEN
     140
169  REM ====================
170  REM = EXTRACT NEXT WORD =
171  REM ====================
177 SS$ =  MID$ (LI$,SB + 1,CP -
     SB - 1)
179  IF SS$ <  > "" THEN NW = NW +
     1:WO$(NW) = SS$
180  IF ES THEN  RETURN
185 SB = CP - 1
189  GOTO 130

]
```

LISTING 9.6 TOKENIZE WITH SENTINEL

```
]LIST

10  DIM WO$(100)
20  PRINT "GIMME A STRING"
25  INPUT LI$
28  GOSUB 100
30  FOR I = 1 TO NW
35  PRINT WO$(I)
40  NEXT I
55  GOTO 20
100  REM ========================
     ===
101  REM = EXTRACT WORDS FROM LIN
     E =
102  REM =
       =
103  REM =  INPUT: LI$ - LINE OF
       =
104  REM =          TEXT.
       =
105  REM = OUTPUT: NW - NUMBER OF
       =
106  REM =          WORDS FOUND.
       =
107  REM =          WO$ - WORDS
       =
108  REM =          FOUND STORED I
     N =
109  REM =          STRING ARRAY.
       =
110  REM =
       =
111  REM ========================
     ===
114  REM ====================
115  REM = INSERT SENTINEL =
116  REM ====================
120 LI$ = LI$ + " "
124  REM ==============
125  REM = INITIALIZE =
126  REM ==============
130 SB = 0:CP = 1:NW = 0
134  REM ========================
     ==
135  REM = EXTRACT NEXT CHARACTER
       =
136  REM ========================
     ==
140 C$ =  MID$ (LI$,CP,1)
144  REM ========================
```

```
    ====
145  REM = CHECK FOR TRAILING BLA
     NK =
146  REM ========================
     ====
150  IF C$ <  > " " THEN CP = CP +
     1: GOTO 140
154  REM ======================
155  REM = EXTRACT NEXT WORD =
156  REM ======================
160 SS$ =  MID$ (LI$,SB + 1,CP -
     SB - 1)
165  IF SS$ <  > "" THEN NW = NW +
     1:WO$(NW) = SS$
169  REM ========================
170  REM = ADVANCE TO NEXT WORD =
171  REM ========================
175 SB = CP
179  REM ========================
     ===
180  REM = CHECK FOR END OF STRIN
     G =
181  REM ========================
     ===
185  IF SB <  LEN (LI$) THEN CP =
     CP + 1: GOTO 140
199  RETURN
```

]&U

Chapter

10

APPLESOFT Trivia

In this chapter we will revisit the APPLE Trivia program of the Integer Basic section. Our aim will be to translate the program into APPLESOFT and make some comparisons.

Listing 10.1 presents the APPLE Trivia Quiz of Chapter 2. It has been paraphrased into APPLESOFT BASIC, but should present itself to a user in a fashion identical to the earlier program. We shall pattern our discussion after that of Chapter 2.

1. USE OF THE SCREEN

With minor differences, all of the screen handling tools of Integer BASIC are also available in APPLESOFT.

Cursor Controls

Integer BASIC	APPLESOFT BASIC
VTAB	**VTAB**
TAB	**HTAB**

ROM Monitor Support

All of these routines are available from APPLESOFT. However, APPLESOFT does *not* allow the user to write:

CALL identifier

Consequently, the user is forced to use (and remember) the actual memory addresses of these routines:

Integer BASIC	APPLESOFT BASIC
CALL HOME	**CALL −936**
CALL CLREOL	**CALL −868**
CALL CLREOP	**CALL −958**

Note that APPLESOFT also provides a statement equivalent in effect to CALL −936; namely, HOME.

Highlighting—Inverse and Flashing Attributes

APPLESOFT allows the user to switch between three varieties of text display:

Normal
Inverse
Flashing

by using single APPLESOFT commands. This is in contrast to the Integer BASIC technique which requires POKEs:

Integer BASIC	APPLESOFT BASIC
POKE 50,63	**INVERSE**
POKE 50,255	**NORMAL**
(No equivalent)	**FLASH**

(With some effort, it is possible to produce consistent flashing displays from Integer BASIC. Compare the author's article "Integer FLASH for the APPLE II" in *MICRO*:37, June 1981.)

In the APPLE Trivia Quiz program, we see the use of INVERSE mode in the scoring subroutine, MARK. Compare the two versions—notice that the APPLESOFT version reads slightly more clearly.

The Scrolling Window

The support of the scrolling window in APPLESOFT involves the same four parameters as used by Integer BASIC. The technique of setting these parameters is identical—new values are stored into the relevant RAM locations using POKEs.

To see what differences there are in the handling of the scrolling window in APPLESOFT, try running the following program. It is the same program used in Chapter 2 to demonstrate Integer BASIC's window handling:

```
 0 TEXT:HOME
 1 VTAB 10:HTAB 10:PRINT "**********"
 2 FOR I=1 TO 10:HTAB 10:PRINT
   "*                  *":NEXT I
 3 HTAB 10:PRINT "**********"
10 POKE 32,10:POKE 33,10
12 POKE 34,10:POKE 35,20
25 VTAB 1:HTAB 5:PRINT "LINE 1 COL 5"
26 VTAB 10:HTAB 25:PRINT "HI"
27 VTAB 23:HTAB 5:PRINT "23,5"
```

You can write similar programs yourself for fully exploring the nature of the scrolling window.

Both Integer BASIC and APPLESOFT provide the TEXT statement for resetting the window to full-screen text mode. That selects all 24 lines and 40 columns for display and scrolling of text. The APPLESOFT HOME command, which is equivalent to CALL −936 in either Integer BASIC or APPLESOFT, has the effect of clearing the screen *inside* the scrolling window and putting the cursor in the upper left-hand corner of the window.

Use BASIC statements instead of POKEs or CALLs

APPLESOFT provides the HOME, NORMAL, INVERSE, and FLASH statements. These may and *should* be used instead of the corresponding CALLs or POKEs, as in Integer BASIC. This will result in programs which are easier to read and understand.

Menus

Almost everything we said in Chapter 2 about menu handling for Integer BASIC applies without change to APPLESOFT. Reread Sections 2 and 3 of Chapter 2 and apply them to the program of Listing 10.1.

2. APPLESOFT PROGRAMMING STYLE

We have attempted to adopt similar styles in coding the Trivia Quiz in the two APPLE BASICs. However, some differences in style are *forced* on us. Let us revisit the programming tips of Chapter 2 and see how they apply in APPLESOFT. At the same time, let us introduce some new tips which are unique to APPLESOFT itself.

Use skeleton programs

The same advice given in Chapter 2 applies to APPLESOFT programs. Almost all BASICs are line-number oriented. This means that careful distribution of line numbers into program sections *ahead of time* can pay dividends later when coding or expanding a program. The skeleton program suggested in Figure 2.2 applies equally well to APPLESOFT programs and has been followed in the implementation of the program of Listing 10.1.

Use COMMENT banners

This tip applies to programs written in *any* language. If anything, our example programs *underuse* this tip. Studying both implementations of APPLE Trivia will show almost identical comment banners.

Use more subroutines

Again, this tip applies to just about any language in which you will ever write programs. It is a well-known psychological principle that the human mind is capable of simultaneously considering only *so much* information. This phenomenon exhibits itself in programming as well as other intellectual disciplines. Here it is especially important to break large problems down into smaller problems. In programming terms: to make big programs, start with small programs or subroutines.

The subroutine structure of Listing 10.1 is almost identical to that of Listing 2.1. As an exercise, construct a diagram analogous to that of Figure 2.3, but for the APPLESOFT version of APPLE Trivia.

Name your subroutines

APPLESOFT unfortunately discourages this coding technique. The APPLESOFT GOSUB statement is limited to the form:

GOSUB <linenumber>

You have to use numbers, and trying to use names will only get you a syntactic slap on the wrist for your trouble. This leads to programs which are far less readable, as pointed out in Chapter 2. There are ways around this, which we now discuss.

Comment your GOSUBs

APPLESOFT, as does Integer BASIC, allows REM statements to appear on the *same* line as other statements. One way to simulate Integer BASICs GOSUB identifier construct is therefore as follows:

- Limit all GOSUBs to a line by themselves.
- Put a REM statement at the end of each GOSUB line, using the *name* of the called subroutine.

For example, from APPLESOFT Trivia, we have:

```
90 GOSUB 31000:REM INIT
91 GOSUB 32000:REM INTRODUCTION
```

Compare this with:

```
90 GOSUB 31000:GOSUB 32000
```

Use a subroutine dictionary

If you read Listing 10.1, you will discover that we have been sloppy. We have not entirely followed our own advice. In many places we have been lazy—merely stringing together a bunch of GOSUBs without commenting them. This has been alleviated somewhat by the inclusion of a *subroutine dictionary:* see lines 10000 to 10135. This is a sequence of comment lines which gives *names* to all the subroutines of the program. Also indicated are the line numbers corresponding to each named subroutine.

In order to make this practice easier to adhere to, we have written a subroutine dictionary generator program. It is shown in Listing 10.2. It takes information provided by the user and creates a text file. The text file contains the BASIC REM statements forming the dictionary. This may be printed out for user cross reference while studying the program. It may also be EXECed into the program itself.

Note that the program uses line numbers beginning with 10000. You will have to *avoid* the use of these if you plan to EXEC the dictionary you create into the program it describes. Note also that the dictionary writer has been used *on itself*. This is definitely a case of practicing what you preach!

Use a declaration section

APPLESOFT limits the usefulness of this tip.It only allows *two character* identifiers. It prohibits the use of identifiers in many places where Integer BASIC allows them. On the other hand, it is still a good idea to gather together all DIM statements and place them in the declaration section.

Study the declaration section of Listing 2.1. Then study Listing 10.1. You'll see that the subroutine dictionary and GOSUB comments take the place of most of the Integer BASIC version. The only two constant declarations which survive in Listing 10.1 are:

```
KB = -16384 : CL = -16368
```

Sampling the keyboard

The same techniques explained in Chapter 2 will work when used in APPLESOFT. In Listing 10.1, this is implemented in the GETKEY subroutine beginning at line 2200.

APPLESOFT also provides a GET statement which allows the input of a single character without requiring the user to push RETURN. The form of the statement is:

```
GET K$
```

and returns the single character typed by the user in the string variable K$. This may be incorporated into APPLESOFT Trivia by replacing line 200 as follows:

```
200 VTAB 14:HTAB 5:CALL -868:GET
    K$:KE= ASC(K$)
```

The value of KE which results in this way will be identical to that returned by the GETKEY subroutine. There is one significant difference in this approach: Using the statement GET K$ will cause a blinking cursor to be displayed on the screen. This difference is slight, but there is an argument for avoiding the appearance of the cursor. Namely, when input is requested by a program and a cursor shows on the display, the psychological tendency on behalf of many users is to hit RETURN *after* pressing a key. This tendency seems to disappear in many cases if there is no cursor blinking away in front of the program user. Of course, even without the cursor, the user might still hit RETURN. Which technique to use must therefore be considered a matter of individual taste.

Protecting against "empty" inputs

It is a good idea to take protective measures in situations such as that just described. Even *if* you sample the keyboard; even *if* you show the user no visible cursor, the user may still generate an extra RETURN. To protect against this occurrence is easy, given the existence of the string function LEN. The expression LEN(A$) provides the length of the string variable A$. The lines of code:

```
100 INPUT A$
105 IF LEN(A$) = 0 THEN 100
```

will protect against the spurious RETURN syndrome.

Use of GOSUB in APPLESOFT

The "unadorned" GOSUB statement in APPLESOFT is limited. You may only refer to a single *fixed* line number. Thus the trick used in Integer BASIC:

```
GOSUB SUBJECTS(TYPE)
```

is not applicable in APPLESOFT. APPLESOFT provides instead the ON...GOSUB statement. This uses the values of an expression to select which of a number of subroutines to invoke. This is standard APPLESOFT and is fully explained in the APPLESOFT Programming Reference Manual published by APPLE Computer Co. We mention it here for contrast with the Integer BASIC coding presented in Chapter 2.

Use of GOTO in APPLESOFT

Just as the coding of GOSUB SUBJECTS(TYPE) was replaced by an ON...GOSUB statement, the coding GOTO 1160+CLASS*10 has been replaced by the statement:

```
ON CL GOTO 1160,1170,1180,1190
```

LISTINGS

LISTING 10.1 APPLESOFT TRIVIA QUIZ

```
]LIST
1  REM ============================
2  REM =                          =
3  REM =   APPLE TRIVIA QUIZ      =
4  REM =            BY            =
5  REM =DR. RICHARD C. VILE, JR.=
6  REM = ALL COMMERCIAL RIGHTS    =
7  REM =       RESERVED           =
8  REM =                          =
9  REM ============================

15 KB =  - 16384:CL =  - 16368
50  DIM O(10),AN(10)
51  DIM TK(5),QU$(10),AN$(10)
90  GOSUB 31000: REM  INIT
91  GOSUB 32000: REM  INTRODUCTIO
    N
100  HOME : NORMAL : VTAB 5: HTAB
    5
105  PRINT "CHOOSE ONE OF THE FOL
    LOWING:"
110  PRINT
120  HTAB 5: PRINT "(A) CHESS HIS
    TORY"
125  HTAB 5: PRINT "(B) OLYMPICS"

130  HTAB 5: PRINT "(C) BOOKS AND
     AUTHORS"
135  HTAB 5: PRINT "(D) COMPUTER
    LORE"
140  HTAB 5: PRINT "(E) GENERAL I
    NFORMATION"
145  HTAB 5: PRINT "(F) EXIT"
195  GOTO 200
199  PRINT  CHR$ (7): REM  BELL
200  VTAB 14: HTAB 5: CALL  - 868
    : GOSUB 2000: REM  GETKEY
205 TY = KE -  ASC ("@")
210  IF (TY < 1) OR (TY > 6) THEN
    199
215  IF TY <  > 6 THEN 250
220  HOME : END
250  IF  NOT TA(TY) THEN 290
255  VTAB 18: CALL  - 868
260  HTAB 1: PRINT "SORRY BUT YOU
     HAVE ALREADY TAKEN THAT"
265  PRINT "QUIZ.  YOU WILL HAVE
    TO TRY ANOTHER"
270  PRINT "SUBJECT."
275  GOSUB 2100: REM  WAIT
280  GOTO 100
290  ON TY GOSUB 4000,5000,6000,7
    000,8000
299  GOTO 100
1000  REM  ======================
    =
1001  REM  = ALPHABET SUBROUTINE
    =
1002  REM  ======================
    =
1005  VTAB BASE: HTAB 20
1010  PRINT "A."
1011  HTAB 20: PRINT "B."
1012  HTAB 20: PRINT "C."
1013  HTAB 20: PRINT "D."
1014  HTAB 20: PRINT "E."
1015  HTAB 20: PRINT "F."
1016  HTAB 20: PRINT "G."
1017  HTAB 20: PRINT "H."
1018  HTAB 20: PRINT "I."
1019  HTAB 20: PRINT "J."
1049  RETURN
1050  REM  ====================
1051  REM  = MIXUP SUBROUTINE =
1052  REM  ====================
1055  FOR I = 1 TO 10:O(I) =  - 1
    : NEXT I
1060  FOR I = 1 TO 10
1065 J = ( RND (1) * 10) + 1
1070  IF O(J) <  >  - 1 THEN 1065

1075 O(J) = I
1080  NEXT I
1099  RETURN
1100  REM  ======================
    ====
1101  REM  = SHOWANSWERS SUBROUTI
    NE =
1102  REM  ======================
    ====
1110  FOR I = 1 TO 10
1115 WH = O(I)
1120  VTAB BA + I - 1: HTAB 23: PRINT
    AN$(WH)
1125  NEXT I
1149  RETURN
1150  REM  ======================
    =====
1151  REM  = GETRESPONSES SUBROUT
    INE =
1152  REM  ======================
    =====
```

```
1154  FOR I = 1 TO 10:AN(I) = 0: NEXT
      I
1155  GOSUB 1300: GOSUB 2000: GOSUB
      1250
1156  ON CL + 1 GOTO 1160,1170,11
      80,1190
1160  REM  *****  CLASS = 0  ****
      *
1162 INDEX = KEY -  ASC ("@"):A$ =
      MID$ (AL$,INDEX,1)
1163  PRINT A$;:ANSWERS(SPOT) = I
      NDEX
1164  GOSUB 1350: GOSUB 1300
1165  GOTO 1155
1170  REM  *****  CLASS = 1  ****
      *
1172  IF KEY = 21 THEN  GOSUB 135
      0
1173  IF KEY = 8 THEN  GOSUB 1400

1175  GOTO 1155
1180  REM  *****  CLASS = 2  ****
      *
1182  GOSUB 1450
1185  RETURN
1190  REM  *****  CLASS = 3  ****
      *
1192  GOSUB 1600
1195  GOTO 1155
1199  RETURN
1200  REM  ==================
1201  REM  = ANSWER OUTLINE =
1202  REM  ==================
1210  VTAB 18: HTAB 1
1215  PRINT "1.      2.      3.
      4.     5.      "
1220  PRINT "6.      7.      8.
      9.     10."
1225  PRINT : PRINT "MOVE CURSOR
      WITH RIGHT AND LEFT ARROWS"
1230  PRINT "HIT ENTER WHEN FINIS
      HED"
1249  RETURN
1250  REM  ======================
      =
1251  REM  = CLASSIFY SUBROUTINE
      =
1252  REM  ======================
      =
1255 CLASS = 3
1260  IF (KEY = 8) OR (KEY = 21) THEN
      CLASS = 1
1265  IF (KEY >  =  ASC ("A")) AND
      (KEY <  =  ASC ("Z")) THEN C
      LASS = 0
1270  IF KEY = 13 THEN CLASS = 2
1299  RETURN
1300  REM  ======================

1301  REM  = POSITION TO ANSWER =

1302  REM  = BLANK SPOT.        =

1303  REM  ======================

1305  GOSUB 1500
1310  VTAB 18 + ROW
1315  HTAB 7 * COL + 3
1320  POKE 50,63: PRINT " ";: POKE
      50,255
1349  RETURN
1350  REM  ===============
1351  REM  =    AHEAD    =
1352  REM  ===============
1355  GOSUB 1450
1356 SPOT = SPOT + 1
1360  IF SPOT = 11 THEN SPOT = 1
1399  RETURN
1400  REM  ===============
1401  REM  =      BACK     =
1402  REM  ===============
1410  GOSUB 1450: REM  ERASEBLOT
1411 SPOT = SPOT - 1
1415  IF SPOT = 0 THEN SPOT = 10
1449  RETURN
1450  REM  ======================
      ==
1451  REM  = ERASEBLOT SUBROUTINE
      =
1452  REM  ======================
      ==
1455  GOSUB 1500: REM  CONVERT
1460  VTAB 18 + ROW
1465  HTAB 7 * COL + 3: PRINT " "
      ;
1499  RETURN
1500  REM  ===============
1501  REM  = CONVERT SPOT =
1502  REM  ===============
1505 ROW = SPOT / 6
1510 T = SPOT - 1
1515 COL = ((T / 5 -  INT (T / 5)
      ) * 5 + .05)
1549  RETURN
1550  REM  ================
1551  REM  = MARK: SCORING =
1552  REM  ================
1555 RI = 0
```

LISTING 10.1 (*cont.*)

```
1560  FOR I = 1 TO 10
1561  IF AN(O(I)) <  > I THEN 156
      5
1562 RI = RI + 1
1563  VTAB BA + O(I) - 1: HTAB 1:
      INVERSE : PRINT O(I);".";
1564  NORMAL
1565  NEXT I
1566  VTAB 21: HTAB 1: CALL  - 86
      8
1567  PRINT : CALL  - 868
1568  PRINT "YOU GOT ";
1570  FOR II = 1 TO RI: GOSUB 160
      0: NEXT II
1572  PRINT RIGHT;" CORRECT."
1599  RETURN
1600  REM  ================
1601  REM  =      TOOT      =
1602  REM  ================
1605  PRINT "";: REM   CONTROL-G
1610  FOR I = 1 TO DE: NEXT I
1649  RETURN
1700  REM  ======================
      ====
1701  REM  = QUIZ - DRIVER CALLED
        =
1702  REM  =         FROM INDIVIDU
      AL =
1703  REM  =         TOPICS SECTIO
      NS =
1704  REM  ======================
      ====
1720  GOSUB 2200
1725  GOSUB 1000
1730  GOSUB 1050
1735  GOSUB 1100: PRINT
1736  PRINT "********************
      ********************"
1740  GOSUB 1200
1742 SPOT = 1: GOSUB 1300
1744  GOSUB 1150: REM  GET RESPON
      SES
1745  GOSUB 1550: REM  MARK
1748  GOSUB 2100: REM  WAIT
1749  RETURN
2000  REM  ======================

2001  REM  = GET KEY SUBROUTINE =

2002  REM  ======================

2005 KE =  PEEK (KB): IF KE < 128
       THEN 2005
2010  POKE  - 16368,0
2015 KE = KE - 128
2049  RETURN
2100  REM  ==================
2101  REM  = WAIT SUBROUTINE =
2102  REM  ==================
2105  POKE 50,63: VTAB 24: HTAB 5
      : PRINT "PRESS ANY KEY TO CO
      NTINUE";
2110  POKE CLR,0
2115  GOSUB 2000: REM  GETKEY
2120  POKE 50,255
2149  RETURN
2200  REM ====================
2201  REM =  PRINT QUESTIONS  =
2202  REM ====================
2205  VTAB BASE
2210  FOR I = 1 TO 10
2215  PRINT QU$(I)
2220  NEXT I
2249  RETURN
4000  REM  ====================
4001  REM  =                    =
4002  REM  =  CHESS HISTORY    =
4003  REM  =                    =
4004  REM  ====================
4010  CALL  - 936: PRINT
4015  PRINT "MATCH THE NAMES OF T
      HE FOLLOWING WORLD"
4016  PRINT "CHESS CHAMPIONS - LA
      ST NAME TO FIRST"
4017  PRINT
4020  GOSUB 4100: GOSUB 4200:BA =
      5
4025  GOSUB 1700: REM  QUIZ
4048 TAKEN(1) = 1
4049  RETURN
4100 QU$(1) = "1. CAPABLANCA"
4105 QU$(2) = "2. MORPHY"
4110 QU$(3) = "3. STEINITZ"
4115 QU$(4) = "4. SPASSKY"
4120 QU$(5) = "5. BOTVINNIK"
4125 QU$(6) = "6. ALEKHINE"
4130 QU$(7) = "7. ANDERSSEN"
4135 QU$(8) = "8. FISCHER"
4140 QU$(9) = "9. PETROSYAN"
4145 QU$(10) = "10.SMYSLOV"
4149  RETURN
4200 AN$(1) = "JOSE RAOUL"
4205 AN$(2) = "PAUL C."
4210 AN$(3) = "WILHELM"
4215 AN$(4) = "BORIS"
4220 AN$(5) = "MIKHAIL"
4225 AN$(6) = "ALEXANDER"
4230 AN$(7) = "ADOLF"
```

```
4235 AN$(8) = "ROBERT"
4240 AN$(9) = "TIGRAN"
4245 AN$(10) = "VASSILY"
4249  RETURN
4999  RETURN
5000  REM  ====================
5001  REM  =                    =
5002  REM  =   OLYMPICS         =
5003  REM  =                    =
5004  REM  ====================
5010  CALL  - 936
5015  PRINT : PRINT "MATCH THE NA
     MES OF THE FOLLOWING "
5016  PRINT "OLYPIANS WITH THEIR
     EVENTS:"
5017  PRINT
5020  GOSUB 5100: GOSUB 5200:BA =
     5
5025  GOSUB 1700: REM  QUIZ
5048 TAKEN(2) = 1
5049  RETURN
5100 QU$(1) = "1. BOB SEAGREN"
5105 QU$(2) = "2. PETER SNELL"
5110 QU$(3) = "3. MARK SPITZ"
5115 QU$(4) = "4. RAFER JOHNSON"
5120 QU$(5) = "5. CAROL HEISS"
5125 QU$(6) = "6. EMIL ZATOPEK"
5130 QU$(7) = "7. VALERY BRUMEL"
5135 QU$(8) = "8. AL OERTER"
5140 QU$(9) = "9. SPYROS LOUES"
5145 QU$(10) = "10.REX CAWLEY"
5149  RETURN
5200 AN$(1) = "POLE VAULT"
5205 AN$(2) = "MILE RUN"
5210 AN$(3) = "SWIMMING"
5215 AN$(4) = "DECATHLON"
5220 AN$(5) = "FIGURE SKATING"
5225 AN$(6) = "5000 METER RUN"
5230 AN$(7) = "HIGH JUMP"
5235 AN$(8) = "DISCUS THROW"
5240 AN$(9) = "MARATHON RUN"
5245 AN$(10) = "400 METER HURDLES
     "
5249  RETURN
5999  RETURN
6000  REM  ====================
6001  REM  =                    =
6002  REM  = BOOKS AND AUTHORS =
6003  REM  =                    =
6004  REM  ====================
6010  CALL  - 936: PRINT
6015  PRINT "MATCH THE NAMES OF T
     HE FOLLOWING AUTHORS";
6016  PRINT "TO THE WORKS WHICH T
     HEY WROTE."
6017  PRINT
6020  GOSUB 6100: GOSUB 6200:BA =
     5
6025  GOSUB 1700: REM  QUIZ
6048 TAKEN(3) = 1
6049  RETURN
6100 QU$(1) = "1. J. CLAVELL"
6105 QU$(2) = "2. P.G. WODEHOUSE"

6110 QU$(3) = "3. J. B. CABELL"
6115 QU$(4) = "4. G. ORWELL"
6120 QU$(5) = "5. C. P. SNOW"
6125 QU$(6) = "6. C. S. LEWIS"
6130 QU$(7) = "7. J. STEINBECK"
6135 QU$(8) = "8. J. DOS PASSOS"
6140 QU$(9) = "9. J. LONDON"
6145 QU$(10) = "10.R. PENN WARREN
     "
6199  RETURN
6200 AN$(1) = "KING RAT"
6205 AN$(2) = "PIGS HAVE WINGS"
6210 AN$(3) = "JURGEN"
6215 AN$(4) = "1984"
6220 AN$(5) = "THE MASTERS"
6225 AN$(6) = "SURPRISED BY JOY"
6230 AN$(7) = "SWEET THURSDAY"
6235 AN$(8) = "MANHATTAN TRANSFER
     "
6240 AN$(9) = "THE SEA WOLF"
6245 AN$(10) = "NIGHT RIDER"
6249  RETURN
7000  REM  ====================
7001  REM  =                    =
7002  REM  =  COMPUTER LORE    =
7003  REM  =                    =
7004  REM  ====================
7010  CALL  - 936: PRINT
7015  PRINT "MATCH THE NAMES OF T
     HE FOLLOWING"
7016  PRINT "COMPUTERS TO THEIR M
     ANUFACTURERS."
7017  PRINT
7020  GOSUB 7100: GOSUB 7200:BASE
      = 5
7025  GOSUB 1700: REM  QUIZE
7048 TAKEN(4) = 1
7049  RETURN
7100 QU$(1) = "1. LEVEL 6"
7105 QU$(2) = "2. NOVA"
7110 QU$(3) = "3. B5500"
7115 QU$(4) = "4. PDP-8"
7120 QU$(5) = "5. SIGMA-7"
7125 QU$(6) = "6. 6600"
```

LISTING 10.1 *(cont.)*

```
7130 QU$(7) = "7. ALTO"
7135 QU$(8) = "8. SPECTRA 70"
7140 QU$(9) = "9. 1130"
7145 QU$(10) = "10.1108"
7149  RETURN
7200 AN$(1) = "HONEYWELL"
7205 AN$(2) = "DATA GENERAL"
7210 AN$(3) = "BURROUGHS"
7215 AN$(4) = "DIGITAL"
7220 AN$(5) = "SDS"
7225 AN$(6) = "CONTROL DATA"
7230 AN$(7) = "XEROX"
7235 AN$(8) = "RCA"
7240 AN$(9) = "IBM"
7245 AN$(10) = "SPERRY UNIVAC"
7249  RETURN
7999  RETURN
8000  REM  ======================
     =
8001  REM  =
     =
8002  REM  = GENERAL INFORMATION
     =
8003  REM  =
     =
8004  REM  ======================
     =
8010  CALL  - 936: PRINT
8899  RETURN
10000  REM ======================
     ============================

10001  REM =
                                 =

10002  REM =    S U B R O U T I N
      E  D I C T I O N A R Y    =

10003  REM =
                                 =

10004  REM =
     ROUTINE NAME          LINE  =

10005  REM =
     ******* ****         *****  =

10006  REM =
                                 =

10010  REM =
        ALPHABET          1000  =

10015  REM =
              MIXUP       1050  =

10020  REM =
     SHOW ANSWERS         1100  =

10025  REM =                    G
     ET RESPONSES         1150  =

10030  REM =               PRINT AN
     SWER OUTLINE         1200  =

10035  REM =                CLASSI
     FY INPUT KEY         1250  =

10040  REM =    POSITION TO ANSWE
     R BLANK SPOT         1300  =

10045  REM =                  MOVE
     CURSOR AHEAD         1350  =

10050  REM =                   MOVE
      CURSOR BACK         1400  =

10055  REM =
     ERASE CURSOR         1450  =

10060  REM =       CONVERT SPOT TO
      ROW AND COL         1500  =

10065  REM =           MARK - SCO
     RING ROUTINE         1550  =

10070  REM =
             TOOT         1600  =

10075  REM =             QUIZ - C
     OMMON DRIVER         1700  =

10080  REM =
          GET KEY         2000  =

10085  REM =
             WAIT         2100  =

10090  REM =                  PRI
     NT QUESTIONS         2200  =

10095  REM =                CHESS
     HISTORY QUIZ         4000  =

10100  REM =                    O
     LYMPICS QUIZ         5000  =

10105  REM =           BOOKS AND
```

```
     AUTHORS QUIZ          6000  =

10110  REM =                 COMPUT
     ER LORE QUIZ          7000  =

10115  REM =           GENERAL INFO
     RMATION QUIZ          8000  =

10120  REM =                   INI
     TIALIZATIONS         31000  =

10125  REM =
     INTRODUCTION          32000  =

10130  REM =
                                  =

10135  REM ======================
     ============================

31000  REM  ====================
31001  REM  = INITIALIZATIONS =
31002  REM  ====================
31010  FOR I = 1 TO 10:O(I) = I: NEXT
     I
31015 AL$ = "ABCDEFGHIJKLMNOPQRST
     UVWXYZ"
31999  RETURN
32000  REM  ====================
32001  REM  =                   =
32002  REM  =   INTRODUCTION    =
32003  REM  =                   =
32004  REM  ====================
32005  CALL  - 936
32010  VTAB 5
32015  PRINT "  WELCOME TO THE AP
     PLE TRIVIA QUIZ!"
32020  PRINT "YOU WILL BE GIVEN A
      CHOICE OF SEVERAL"
32025  PRINT "TOPICS TO BE QUIZZE
     D ON.  EACH TOPIC"
32030  PRINT "WILL HAVE A TEN QUE
     STION QUIZ OF ONE"
32035  PRINT "OF THE FOLLOWING TY
     PES:"
32037  PRINT
32040  PRINT "      1. MATCHING"
32045  PRINT "      2. MULTIPLE CH
     OICE"
32050  PRINT "      3. SHORT ANSWE
     R"
32055  PRINT "      4. TRUE OR FAL
     SE"
32060  PRINT : PRINT "  YOU MAY T
     AKE ALL THE TIME YOU WISH"
32065  PRINT "TO ANSWER EACH QUIZ
     , BUT ONCE A GIVEN"
32070  PRINT "TOPIC HAS BEEN TAKE
     N, YOU MAY NOT "
32075  PRINT "RETURN TO IT."
32080  PRINT "GOOD LUCK!!!"
32095  GOSUB 2100: REM  WAIT
32099  RETURN

]
```

LISTING 10.2 SUBROUTINE DICTIONARY WRITER

```
]LIST

1  REM  ==========================
2  REM  =       APPLESOFT        =
3  REM  = SUBROUTINE DICTIONARY  =
4  REM  =        WRITER          =
5  REM  =                        =
6  REM  =DR. RICHARD C. VILE, JR.=
7  REM  = ALL COMMERCIAL RIGHTS  =
8  REM  =        RESERVED        =
9  REM  ==========================
10  DIM SN$(200),LN(200)
11 D$ =  CHR$ (4): REM  CONTROL-D
100  GOSUB 9000: REM  INTRODUCTION
102  GOSUB 5000: REM  INIT STRINGS
105  HOME : VTAB 5: HTAB 5
110  PRINT "CHOOSE THE MODE OF OPERATION:": PRINT
115  HTAB 5: PRINT  CHR$ (219);"A] INTERACTIVE INPUT"
```

LISTING 10.2 (*cont.*)

```
120  HTAB 5: PRINT  CHR$ (219);"B] INPUT FROM FILE"
125  HTAB 5: PRINT  CHR$ (219);"C] EXIT"
150  PRINT
155  GET AN$:WH =  ASC (AN$) -  ASC ("@")
160  IF WH =  - 61 THEN  HOME : END
165  IF WH < 0 THEN 155
170  ON WH GOSUB 200,250,999
175  GOSUB 1000: REM  COLLECT INPUT
180  GOSUB 1100: REM  SORT SUBROUTINE NAMES
185  GOSUB 1200: REM  OUTPUT DICTIONARY
190  GOTO 105
200  REM ================================
201  REM = SET UP FOR INTERACTIVE INPUT =
202  REM ================================
205 FI = 0
249  RETURN
250  REM ============================
251  REM = SET UP FOR FILE INPUT =
252  REM ============================
260  HOME : VTAB 5: HTAB 5
265  INPUT "NAME OF INPUT FILE===>";SF$
270  PRINT D$
275  PRINT D$;"OPEN ";SF$
280 FI = 1
299  RETURN
999  HOME : END
1000  REM ================
1001  REM = COLLECT INPUT =
1002  REM ================
1010 I = 0
1015  PRINT "INPUT NAME,LINE NUMBER PAIRS": PRINT : PRINT "INPUT 0
     ,0 TO QUIT"
1050 I = I + 1
1055  IF FI THEN  GOSUB 1650: REM  READ FROM FILE
1062  INPUT SN$(I),LN(I)
1063  IF  LEN (SN$(I)) > 40 THEN SN$(I) =  LEFT$ (SN$(I),40)
1065  IF LN(I) > 0 THEN 1050
1070 NN = I - 1: REM  NUMBER OF SUBROUTINE NAMES
1075  IF FI THEN  PRINT D$;"CLOSE ";SF$
1099  RETURN
1100  REM ============================
1101  REM = SORT SUBROUTINE NAMES =
1102  REM ============================
1103  PRINT "SORTING"
1105  FOR I = 1 TO NN - 1
1110  FOR J = 1 TO NN - I
1120  IF LN(J) <  = LN(J + 1) THEN 1150
1130 T$ = SN$(J):T = LN(J)
1135 SN$(J) = SN$(J + 1):LN(J) = LN(J + 1)
1140 SN$(J + 1) = T$:LN(J + 1) = T
1150  NEXT J
1155  NEXT I
1160  PRINT "DONE SORTING"
```

```
1199  RETURN
1200  REM ======================
1201  REM = OUTPUT DICTIONARY =
1202  REM ======================
1205  INPUT "OUTPUT FILE NAME===>";FO$
1210  PRINT D$;"OPEN ";FO$
1211  PRINT D$;"DELETE ";FO$
1212  PRINT D$;"OPEN ";FO$
1215  PRINT D$;"WRITE ";FO$
1220  PRINT "10000REM";HD$
1221  PRINT "10001REM";SP$
1222  PRINT "10002REM";T1$
1223  PRINT "10003REM";SP$
1224  PRINT "10004REM";T2$
1225  PRINT "10005REM";T3$
1226  PRINT "10006REM";SP$
1230  FOR I = 1 TO NN
1235  GOSUB 1300: REM  NAME PADDING
1240  GOSUB 1400: REM   LINE NUMBER PADDING
1245  GOSUB 1600: REM  SELECT FILE OUTPUT
1250  PRINT 10005 + I * 5;
1251  PRINT "REM=  ";
1252  PRINT NP$;SN$(I);
1253  PRINT "         ";
1254  PRINT LP$;LN(I);"  ="
1265  NEXT I
1266  PRINT D$
1270  GOSUB 1600
1275  PRINT 10005 + I * 5;"REM";SP$
1276  PRINT 10005 + (I + 1) * 5;"REM";HD$
1280  PRINT D$;"CLOSE ";FO$
1299  RETURN
1300  REM ================
1301  REM = NAME PADDING =
1302  REM ================
1305 NL =  LEN (SN$(I))
1310 NP$ =  LEFT$ (PD$,31 - NL)
1349  RETURN
1400  REM ========================
1401  REM = LINE NUMBER PADDING =
1402  REM ========================
1405 LP$ = ""
1410  IF LN(I) > 9999 THEN  RETURN
1415 LP$ = LP$ + " "
1420  IF LN(I) > 999 THEN  RETURN
1425 LP$ = LP$ + " "
1430  IF LN(I) > 99 THEN  RETURN
1435 LP$ = LP$ + " "
1440  IF LN(I) > 9 THEN  RETURN
1445 LP$ = LP$ + " "
1449  RETURN
1500  REM ====================
1501  REM =        WAIT        =
```

LISTING 10.2 (*cont.*)

```
1502  REM ====================
1505  VTAB 24: INVERSE : HTAB 5
1510  PRINT "PRESS ANY KEY TO CONTINUE";
1525  POKE  - 16368,0
1530  IF  PEEK ( - 16384) < 128 THEN 1530
1535  POKE  - 16368,0
1540  NORMAL
1549  RETURN
1600  REM ============================
1601  REM =  WRITE TO OUTPUT FILE  =
1602  REM ============================
1605  PRINT D$
1610  PRINT D$;"WRITE ";FO$
1649  RETURN
1650  REM ====================
1651  REM =  READ FROM FILE  =
1652  REM ====================
1659  PRINT D$.
1660  PRINT D$;"READ ";SF$
1699  RETURN
5000  REM ==================
5001  REM =  INIT STRINGS  =
5002  REM =                =
5003  REM = SET UP STRINGS =
5004  REM = FOR TITLE LINES=
5005  REM ==================
5010 EQ$ = "==========="
5011 LE$ = "=     "
5012 RI$ = "     ="
5013 BL$ = "           "
5014 SR$ = "S U B R O U T I N E"
5015 DI$ = "D I C T I O N A R Y"
5016 RN$ = "ROUTINE NAME"
5017 S1$ = "******* ****"
5018 PD$ = BL$ + BL$ + BL$ + " ": REM   31 BLANKS
5019 HD$ = EQ$ + EQ$ + EQ$ + EQ$ + EQ$
5020 SP$ = LE$ + BL$ + BL$ + BL$ + BL$ + RI$
5021 T1$ = LE$ + SR$ + "  " + DI$ + RI$
5022 T2$ = LE$ + BL$ + "       " + RN$ + "          " + "LINE  ="
5023 T3$ = LE$ + BL$ + "       " + S1$ + "         " + "*****  ="
5049  RETURN
9000  REM ==================
9001  REM =  INTRODUCTION  =
9002  REM ==================
9005  HOME : VTAB 5: PRINT "WELCOME TO THE APPLESOFT SUBROUTINE"
9007  PRINT "DICTIONARY WRITER PROGRAM.  THIS PROGRAM";
9009  PRINT "WILL CREATE AN INTERNAL SUBROUTINE"
9011  PRINT "DICTIONARY CONSISTING OF NAMES CHOSEN"
9013  PRINT "BY YOU FOR EACH SUBROUTINE IN A GIVEN"
9015  PRINT "PROGRAM.  EACH SUBROUTINE AND ITS "
9017  PRINT "CORRESPONDING LINE NUMBER WILL BE "
9019  PRINT "LISTED IN THE DICTIONARY, WHICH WILL"
9021  PRINT "BE DUMPED TO AN EXEC FILE.  THE EXEC"
```

```
9023  PRINT "MAY BE USED TO INCORPORATE THE DICT-"
9025  PRINT "IONARY INTO THE PROGRAM IT DOCUMENTS."
9095  GOSUB 1500: REM  WAIT
9099  RETURN
10000  REM ================================================
10001  REM =                                              =
10002  REM =    S U B R O U T I N E  D I C T I O N A R Y   =
10003  REM =                                              =
10004  REM =                      ROUTINE NAME      LINE  =
10005  REM =                      ******* ****      *****  =
10006  REM =                                              =
10010  REM =                 INTERACTIVE INPUT       200  =
10015  REM =                      FILE INPUT         250  =
10020  REM =                   COLLECT INPUT         1000  =
10025  REM =     SORT SUBROUTINE LINE NUMBERS        1100  =
10030  REM =        OUTPUT DICTIONARY TO FILE        1200  =
10035  REM =              LINE NUMBER PADDING        1400  =
10040  REM =                             WAIT        1500  =
10045  REM =                SELECT FILE WRITE        1600  =
10050  REM =                 SELECT FILE READ        1650  =
10055  REM =   STRING CONSTANT INITIALIZATION        5000  =
10060  REM =                      INTRODUCTION        9000  =
10065  REM =                                              =
10070  REM ================================================

]
```

Chapter

11

APPLESOFT Low-Resolution Graphics Techniques

In this chapter we revisit some of the topics introduced in Chapter 3. In particular, we translate the Giant Letters Screen program into APPLESOFT. The abstract techniques discussed in Chapter 3 apply equally well in APPLESOFT, so a review might be in order before proceeding.

We also present a second application of the format string interpreter presented in Chapter 3. This time, we implement the interpreter via a machine-language program. The format strings themselves still appear in the APPLESOFT program, however! Wonder how that's done? Read on.

1. THE GIANT LETTERS SCREEN REVISITED

The program of Listing 11.1 is a direct paraphrasing into APPLESOFT of the program of Listing 3.2. Many points made in Chapter 10 about the difference in style between Integer BASIC and APPLESOFT BASIC become evident here as well. The beauty and usefulness of Integer BASIC's GOSUB expression statement become painfully obvious when we compare lines 222 to 250 of the APPLESOFT version to lines 225 to 250 of the Integer version. The invocation of the picture drawing subroutines is much more flexible in the Integer version. APPLESOFT's ON...GOSUB can be made to serve the same purpose, but it is much more cumbersome and unnatural.

2. THE AMPERSAND COMMAND AND A REVISED GIANT LETTERS

In Chapter 3 we invented a notation which we dubbed "format strings." It enabled us to represent sequences of low-resolution graphics commands compactly. We implemented an interpreter for such strings in Integer BASIC. While this did provide more compact coding, it was a little slow. Well, let's face facts: it was a real dog. It would be more tempting to use the format strings approach if we could speed things up a bit. We have just the ticket in APPLESOFT. There is a way to pass information, such as format strings, from an APPLESOFT program to a machine-language support program.

We will make use of this feature, known as the ampersand command, in order to use a machine-language interpreter for our format strings. We present the interpreter, fully explained, in a later chapter.

The Ampersand Command

The command is mentioned only briefly in the APPLESOFT reference manual. In Appendix G we read:

> The ampersand (&) is intended for the computer's internal use only; it is *not* a proper APPLESOFT command. This symbol, when executed as an instruction, causes an unconditional jump to location $3F5. Use reset ctrl c return to recover.

Well! "internal use only," not "proper," etc. APPLE II users take such advice as a challenge. Rather than heed these rather negative recommendations, many APPLE users have delved into the inner workings of ampersand. In fact, a whole subculture of "amper" routines has been created with many attendant benefits to APPLESOFT users. We shall take this opportunity to explain the ampersand command and add our humble contribution to the lore.

The unconditional jump to location $3F5 allows the user in turn to place an unconditional jump to any machine-language routine deemed useful under the circumstances. That routine may then "look back" into the APPLESOFT program and take further instructions from that source. As it goes along, it may advance pointers into the program in such a way that when it (the machine-language program) returns, APPLESOFT will continue at the next statement. APPLESOFT will be none the wiser for the interruption. How all this is accomplished will be made clear in Chapter 19. For now, let us see how to apply it to the Giant Letters program.

Passing Format Strings to Machine Language

We wish to be able to feed format strings to a machine-language interpreter. The most obvious way to do this would seem to be as follows: In several statements of our APPLESOFT program, we will write an &, followed by a format string:

```
100 &"H050V050C7P55"
```

In fact, this is perfectly feasible. It is not the best method, however. Embedding the strings directly in the APPLESOFT program as individual "statements" means that those statements must be invoked. This would be by normal program control flow—such as ON...GOSUBs. Of course, this poses the same awkwardness as that evidenced by Listing 11.1.

A better way is to pass the *name* of a string variable to the machine-language program, instead of the string itself. For example,

```
100 &B$      or      100 &L$(I)
```

This approach allows much more flexibility. The specific string to be passed may be computed by any algorithm the user desires. The desired string may then be assigned to a string variable and passed on via ampersand.

The Amper-Letters Program

The program of Listing 11.2 takes the approach just described. It reads in a file of format strings representing the giant letters "character set." It then passes these to the format string interpreter, based on the keys struck by the user. The data file read by the program is shown in Listing 11.3.

The machine-language interpreter is invoked at various places; specifically in lines 250, 616, 660, and 667. For example, the "call" in line 677 passes a string:

```
"P05P45V151V153V062"
```

which causes a picture of a bell to be drawn. This is done whenever the user types the Control-G or "bell" character from the keyboard. The "call" in line 250:

```
250 &L(C)
```

is used to pass a selected string from array L to the interpreter. The string passed corresponds to the key that the user strikes. The single statement in line 250 suffices to pass all 94 possibilities. Quite a contrast to lines 222 to 250 of Listing 11.1!

The Amper Letters program is much faster than the analogous part of Listing 3.3. It won't keep up with a speedy touch typist, but is more than adequate for most children and other likely users of such a program.

Amper Applications

In order to run the Amper Letters program, or any other program which uses the machine-language format strings interpreter of Chapter 19, it is first necessary to load the machine-language program and otherwise set up an appropriate environment. We suggest the use of an EXEC file for this purpose. The following is typical (for more information, see Chapter 19):

```
BLOAD AMPER-GRAPHICS.OBJ0
BLOAD AMPER-GRAPHICS.OBJ1
HIMEM:36864
```

Follow this, of course, by a RUN command invoking your APPLESOFT program. For example:

RUN AMPER LETTERS

In creating your own applications using Amper-Graphics, Tables 11.1 and 11.2 may be useful. They give numeric values to use to represent color values (for the format string C command) and the numeric equivalents of the allowable arguments to the other commands. For example,

HAFR

is equivalent to

HLIN COL+10,COL+15 AT ROW+27

Note that the machine-language interpreter expects the values of ROW and COL to be stored in RAM locations 253 and 254, respectively.

TABLE 11.1 Color Codes for Amper Graphics

Black	0	Brown	8
Magenta	1	Orange	9
Darkblue	2	Grey	A
Purple	3	Pink	B
Darkgreen	4	Green	C
Grey	5	Yellow	D
Mediumblue	6	Aqua	E
Lightblue	7	White	F

TABLE 11.2 Amper Graphics: Numeric equivalents of letter arguments

A	10	K	20	U	30
B	11	L	21	V	31
C	12	M	22	W	32
D	13	N	23	X	33
E	14	O	24	Y	34
F	15	P	25	Z	35
G	16	Q	26		
H	17	R	27		
I	18	S	28		
J	19	T	29		

3. EXPLORATIONS

- Design alternate character sets to be used with the Amper Letters program.
- Design truly "giant" letters to be used with Amper Letters. Make them large enough to fill the screen.
- Implement other programs which use the Amper-Strings interpreter.

LISTINGS

LISTING 11.1 GIANT LETTERS IN APPLESOFT

```
]LIST
1  REM  ========================
     =
2  REM  =
     =
3  REM  =  GIANT LETTERS SCREEN
     =
4  REM  =
     =
5  REM  =DR. RICHARD C. VILE, JR.
     =
6  REM  = ALL COMMERCIAL RIGHTS
     =
7  REM  =        RESERVED
     =
8  REM  =
     =
9  REM  ========================
     =
15 KB =  - 16384:CL =  - 16368
16 FU =  - 16302
200  REM  ==================
201  REM  =  MAIN PROGRAM  =
202  REM  ==================
205  GR : PRINT : PRINT : PRINT
208 ROW = 0:COL = 0
210  GOSUB 1000: REM  SET FULL SC
     REEN GRAPHICS
212  GOSUB 1050: REM  FULL WINDOW
215 CC = 7
216  COLOR= CC
218  GOSUB 600: REM  DRIVER SUBRO
     UTINE
220  COLOR= CC:C = C - 32
222  IF C < 0 THEN 270
225  IF C > 9 THEN 230
227  ON C + 1 GOSUB 5000,5025,505
     0,5075,5100,5125,5150,5175,5
     200,5225
228  GOTO 270
230  IF C > 19 THEN 235
231  ON C - 9 GOSUB 5250,5275,530
     0,5325,5350,5375,5400,5425,5
     450,5475
232  GOTO 270
235  IF C > 29 THEN 240
236  ON C - 19 GOSUB 5500,5525,55
     50,5575,5600,5625,5650,5675,
     5700,5725
237  GOTO 270
240  IF C > 39 THEN 245
241  ON C - 29 GOSUB 5750,5775,58
     00,5825,5850,5875,5900,5925,
     5950,5975
242  GOTO 270
245  IF C > 49 THEN 250
246  ON C - 39 GOSUB 6000,6025,60
     50,6075,6100,6125,6150,6175,
     6200,6225
247  GOTO 270
250  ON C - 49 GOSUB 6250,6275,63
     00,6325,6350,6375,6400,6425,
     6450,6475,6500,6525,6550
270 COL = COL + 6: IF COL <  = 35
     THEN 215
272 COL = 0:ROW = ROW + 8
274  IF ROW <  = 40 THEN 215
275  GOSUB 1100: REM  SCROLL
280  GOTO 215
600  REM  ====================
601  REM  =  KEYBOARD DRIVER  =
602  REM  ====================
605  GOSUB 700: REM  GET KEY
610 C = KEY - 128
612  IF C <  > 27 THEN 618
614 ROW = 0:COL = 0
615  COLOR= 0: FOR J = 0 TO 39: VLIN
     0,47 AT J: NEXT J: COLOR= CC

616  GOTO 600
618  IF C = 13 THEN 685
619  IF C = 7 THEN 675
620  IF (C <  > 8 AND C <  > 21) THEN
      RETURN
622  IF C <  > 21 THEN 625
623 C = 32: RETURN
625 COL = COL - 6: IF COL >  = 0 THEN
     650
630 COL = 30:ROW = ROW - 8: IF RO
     W >  = 0 THEN 650
635 COL = 0:ROW = 0
650  COLOR= 0
655  FOR J = 0 TO 6
660  HLIN COL,COL + 4 AT ROW + J
665  NEXT J
670  COLOR= CC: GOTO 600
675  PLOT COL,ROW + 5: PLOT COL +
     4,ROW + 5
677  VLIN ROW + 1,ROW + 5 AT COL +
     1
679  VLIN ROW + 1,ROW + 5 AT COL +
     3
680  VLIN ROW,ROW + 6 AT COL + 2
681  PRINT "";: RETURN
685 ROW = ROW + 8: IF ROW >  = 40
      THEN  GOSUB 1100: REM  SCRO
     LL
687 COL = 0: GOTO 600
700  REM  ==================
```

LISTING 11.1 *(cont.)*

```
701  REM =      GETKEY      =
702  REM = USED TO AVOID   =
703  REM = CURSOR BLOB.    =
704  REM ==================
715 KEY =  PEEK (KB): IF KEY < 12
    8 THEN 715
720  POKE CL,0
749  RETURN
1000  REM  ======================
    ======
1001  REM  = SET FULL SCREEN GRAP
    HICS =
1002  REM  ======================
    ======
1005  POKE FU,0
1010  COLOR= 0
1015  FOR I = 40 TO 47: HLIN 0,39
    AT I: NEXT I
1049  RETURN
1050  REM  ======================
    ====
1051  REM  =      FULLWINDOW
        =
1052  REM  =
        =
1053  REM  =  SET GRAPHICS WINDOW
    TO=
1054  REM  = FULL SCREEN TO ALLOW
        =
1055  REM  = SCROLLING LOW-RES.
        =
1056  REM  ======================
    ====
1060  POKE 32,0: POKE 33,40
1061  POKE 34,0: POKE 35,24
1099  RETURN
1100  REM  ======================
    ====
1101  REM  =        SCROLL
        =
1102  REM  ======================
    ====
1105 SAVCOL =  PEEK (48): COLOR=
    0
1110  FOR J = 1 TO 4
1115  CALL  - 912
1120  HLIN 0,39 AT 46: HLIN 0,39 AT
    47
1125  NEXT J
1130 ROW = 40:COL = 0
1135  COLOR= SAVCOL
1149  RETURN
1150  REM  ==================
1151  REM  =      GETKEY      =
1152  REM  ==================
1155 KEY =  PEEK (KBD): IF KEY <
    128 THEN 1155
1160  POKE CLR,0
5000  REM  ==================
5001  REM  =      SPACE       =
5002  REM  ==================
5024  RETURN
5025  REM  ==================
5026  REM  =  EXCLAMATION   =
5027  REM  ==================
5030  VLIN ROW,ROW + 4 AT COL + 3

5032  PLOT COL + 3,ROW + 6
5049  RETURN
5050  REM  ==================
5051  REM  = DOUBLE QUOTE =
5052  REM  ==================
5060  VLIN ROW,ROW + 1 AT COL + 1

5065  VLIN ROW,ROW + 1 AT COL + 3

5074  RETURN
5075  REM  ==================
5076  REM  =  POUND SIGN  =
5077  REM  ==================
5080  VLIN ROW + 1,ROW + 5 AT COL
    + 1
5082  VLIN ROW + 1,ROW + 5 AT COL
    + 3
5085  HLIN COL,COL + 4 AT ROW + 2

5087  HLIN COL,COL + 4 AT ROW + 4

5099  RETURN
5100  REM ==================
5101  REM =  DOLLAR SIGN  =
5102  REM ==================
5110  VLIN ROW,ROW + 6 AT COL + 2

5111  HLIN COL,COL + 4 AT ROW + 1

5112  HLIN COL,COL + 4 AT ROW + 3

5113  HLIN COL,COL + 4 AT ROW + 5

5114  PLOT COL,ROW + 2: PLOT COL +
    4,ROW + 4
5124  RETURN
5125  REM ==================
5126  REM = PERCENT SIGN =
5127  REM ==================
5130  HLIN COL,COL + 1 AT ROW: HLIN
    COL,COL + 1 AT ROW + 1
```

```
5131  HLIN COL + 3,COL + 4 AT ROW
      + 5: HLIN COL + 3,COL + 4 AT
      ROW + 6
5132  PLOT COL + 4,ROW + 1: PLOT
      COL + 3,ROW + 2
5133  PLOT COL + 2,ROW + 3: PLOT
      COL + 1,ROW + 4
5134  PLOT COL,ROW + 5
5149  RETURN
5150  REM ============
5151  REM = AMPERSAND =
5152  REM ============
5155  PLOT COL + 1,ROW: PLOT COL,
      ROW + 1: PLOT COL + 2,ROW +
      1: PLOT COL + 1,ROW + 2
5156  HLIN COL,COL + 3 AT ROW + 3
      : HLIN COL,COL + 3 AT ROW +
      6
5157  VLIN ROW + 4,ROW + 5 AT COL
      : VLIN ROW + 4,ROW + 5 AT CO
      L + 3
5158  PLOT COL + 4,ROW + 2: PLOT
      COL + 4,ROW + 5
5174  RETURN
5175  REM ===============
5176  REM = SINGLE QUOTE =
5177  REM ===============
5180  HLIN COL + 2,COL + 3 AT ROW

5181  HLIN COL + 2,COL + 3 AT ROW
      + 1
5182  HLIN COL + 2,COL + 3 AT ROW
      + 3
5185  PLOT COL + 3,ROW + 2
5199  RETURN
5200  REM ====================
5201  REM = LEFT PARENTHESIS =
5202  REM ====================
5205  PLOT COL + 2,ROW: PLOT COL +
      2,ROW + 6
5206  PLOT COL + 1,ROW + 1: PLOT
      COL + 1,ROW + 5
5210  VLIN ROW + 2,ROW + 4 AT COL

5224  RETURN
5225  REM =====================
5226  REM = RIGHT PARENTHESIS =
5227  REM =====================
5230  VLIN ROW + 2,ROW + 4 AT COL
      + 4
5232  PLOT COL + 2,ROW: PLOT COL +
      2,ROW + 6
5235  PLOT COL + 3,ROW + 1: PLOT
      COL + 3,ROW + 5
```

LISTING 11.2 AMPER LETTERS

```
]LIST

1   REM  ************************
       *
2   REM  *
       *
3   REM  *      AMPER LETTERS
       *
4   REM  *
       *
5   REM  *DR. RICHARD C. VILE, JR.
       *
6   REM  * ALL COMMERCIAL RIGHTS
       *
7   REM  *        RESERVED
       *
8   REM  *
       *
9   REM  ************************
       *
10 D$ =  CHR$ (4): REM CONTROL-D
15  DIM L$(94): REM  LETTERS ARRA
      Y
20 F$ = "GIANT LETTERS DATA"
25  PRINT D$;"OPEN ";F$
26  PRINT D$;"READ ";F$
28  FOR I = 1 TO 94: INPUT L$(I):
      NEXT I
30  PRINT D$;"CLOSE ";F$
99  POKE 208,0: POKE 216,0
200  REM  ==================
201  REM  =  MAIN PROGRAM  =
202  REM  ==================
205  GR : PRINT : PRINT : PRINT
208  POKE 253,0: POKE 254,0
210  GOSUB 1000: REM  SET FULL SC
     REEN GR
212  GOSUB 1050: REM  FULL WINDOW

215 CC =  INT ( RND (CC + 1) * 15
     ) + 1
216  COLOR= CC
218  GOSUB 600: REM  DRIVER SUBRO
     UTINE
220  COLOR= CC
250  & L$(C)
```

LISTING 11.2 *(cont.)*

```
270  GOSUB 300: REM  ADJUST ROW,C
     OL
275  IF  PEEK (253) < 40 THEN 215

280  GOTO 215
300  REM ======================
301  REM = ADJUST ROW AND COL =
302  REM ======================
305 NR =  PEEK (253):NC =  PEEK (
     254)
310 NC = NC + 6: IF NC < 36 THEN
     350
315 NC = 0:NR = NR + 8: IF NR <  =
     40 THEN 350
320  GOSUB 1100: RETURN
350  POKE 253,NR: POKE 254,NC
399  RETURN
600  REM  ====================
601  REM  =  KEYBOARD DRIVER  =
602  REM  ====================
605  GOSUB 700
610 C = KEY - 128
612  IF C <  > 27 THEN 618
614  POKE 253,0: POKE 254,0
615 B$ = "COF": & B$
616  GOTO 600
618  IF C = 13 THEN 685
619  IF C = 7 THEN 675
620  IF (C <  > 8 AND C <  > 21) THEN
      RETURN
622  IF C <  > 21 THEN 625
623 C = 32: RETURN
625 NC =  PEEK (254) - 6: IF NC >
     = 0 THEN 650
630 NC = 30:NR =  PEEK (253) - 8:
     IF NR >  = 0 THEN 650
635 NC = 0:NR = 0
650  POKE 253,NR: POKE 254,NC
655 B$ = "COH040H041H042H043H044H
     045H046"
660  & B$
665  COLOR= CC: GOTO 600
675 B$ = "P05P45V151V153V062"
677  & B$
679  PRINT  CHR$ (7);
680  GOSUB 300: GOTO 600
685 NR =  PEEK (253) + 8: IF NR >
     = 40 THEN 690
687  POKE 254,0: POKE 253,NR: GOTO
     600
690  GOSUB 1100: POKE 253,40: POKE
     254,0
691  GOTO 600
700  REM ================
701  REM =    GETKEY     =
702  REM = USED TO AVOID =
703  REM = CURSOR BLOB.  =
704  REM ================
715 KEY =  PEEK ( - 16384): IF KE
     Y < 128 THEN 715
720  POKE  - 16368,0
749  RETURN
1000  REM  ======================
     ======
1001  REM  = SET FULL SCREEN GRAP
     HICS =
1002  REM  ======================
     ======
1005  POKE  - 16302,0
1010  COLOR= 0
1015  FOR I = 40 TO 47: HLIN 0,39
     AT I: NEXT I
1049  RETURN
1050  REM  ======================
     ====
1051  REM  =     FULLWINDOW
       =
1052  REM  =
       =
1053  REM  =  SET GRAPHICS WINDOW
     TO=
1054  REM  = FULL SCREEN TO ALLOW
       =
1055  REM  = SCROLLING LOW-RES.
       =
1056  REM  ======================
     ====
1060  POKE 32,0: POKE 33,40
1061  POKE 34,0: POKE 35,24
1099  RETURN
1100  REM  ======================
     ====
1101  REM  =       SCROLL
       =
1102  REM  ======================
     ====
1105 SAVCOL =  PEEK (48): COLOR=
     0
1110  FOR J = 1 TO 4
1115  CALL  - 912
1120  HLIN 0,39 AT 46: HLIN 0,39 AT
     47
1125  NEXT J
1130  POKE 253,40: POKE 254,0
1135  COLOR= SAVCOL
1149  RETURN

]
```

LISTING 11.3 AMPER-LETTERS PATTERN

```
 1      P42H343H244H145H046
 2      P20H131H042P23H134H045P26
 3      H050V050H055V055
 4      P00P20P40P11P31P02P22P42P13P33P04P24P44P15P35P06P26P46
 5      V121V123V451V453
 6      V061H031H043H055
 7      H050V050
 8      P21H132H043H134P25
 9      B0022B0322B2022B2322B5022B5322
10      V062H043B0022
11      V010V012V014H042H044V560V562V564
12      V062V063H010H013H016H340H343H346
13      V061V063H010H013H016H340H343H346
14      H040H043H046V060V062V064
15      P20H131H132H043H044H135H136
16      V062H043
17      H011H012H043H044P05P45
18      H042H043V450V454
19      V060V064V242P10P30P16P36
20      H010V011H121V121
21      H010V011H121V122H232V233H343V344
22      H010H011H340H341H015H016H345H346H132H133H134
23      H043P21P25
24      H130H131P22H043P24P15P35P06P46
25      V040V044V151V153V262
26      D502D205D014P40
27      H050
28      H050
29      H050
30      H050
31      H050
32
33      V042P26
34      V011V013
35      H042H044V151V153
36      V062H041H043H045P02P44
37      P11P05P14P23P32P41P35
38      P10V120V122P13V460V452P16P34P43
39      P20P11P02
40      P20P11V240P15P26
41      P20P31V244P35P26
42      V062P01P12P32P41P05P14P34P45
43      H133V242
44      V562P16
45      H133
46      H125H126
47      P41P32P23P14P05
48      H136H130V154V150
49      H120P01V062H046
50      H130P01V124H133V460H046
51      H130P01V124H233V454P05H136
52      V063H044P21P12P03
53      H040V030H033V454H136P05
54      H130P41V150H133H136V454
55      H040V124P33P24P15P06
56      H130H133H136V120V124V450V454
```

LISTING 11.3 (*cont.*)

```
57      H130H133H136V154V120P05
58      P22P24
59      P23V562P16
60      P30P21P12P03P14P25P36
61      H132H134
62      D014P34P25P16
63      H040H243P01V034P24P26
64      V060V034V564H040H046H243P22P45
65      P20P11P31V260V264H044
66      H030H033H036V060V124V454
67      H040H046V060
68      H030H036V060V154
69      H040H046V060H023
70      H040V060H023
71      H030H036V060V463H244
72      V060V064H043
73      H130H136V062
74      H140V063H136P05
75      V060P40P31P22P13P24P35P46
76      V060H046
77      V060V064P11P22P31
78      V060V064P12P23P34
79      V060V064H040H046
80      V060V034H040H043
81      H030H046V060V063
82      H030H033V060V124P24P35P46
83      H040H043H046V030V364P41P05
84      H040V062
85      V060V064H046
86      V020V024V341V343V562
87      V060V064V342P15P35
88      V010V014V560V564P12P32P23P14P34
89      V010V014V362P12P32
90      H040H046P05P14P23P32P41
91      H020V060H026
92      H050
93      H230H236V063
94      P22P13P33P04P44
```

Chapter

12

APPLE Pascal: Features and Use

Pascal is a relatively new language on the computer scene and has not gained the same degree of popularity as BASIC, especially in the microcomputer world. We hope to demonstrate some of the fascinating power and naturalness of the language in this section of the book. We begin by presenting a precis of the features of the language, with emphasis on those which set it apart from BASIC and make it appealing to use.

1. PROGRAM STRUCTURE

Some of the immediately striking differences between Pascal and BASIC, aside from Pascal's lack of line numbers, are the additional structuring mechanisms available in the language. In particular, it is almost impossible to write a sophisticated Pascal program without using

- Procedures
- Functions

The skeleton of a Pascal program looks roughly like this:

```
program skeleton
const
    Declaration of constants used by the program
type
    Declaration of user-defined data types
var
    Declaration of variables used in the program
    "Declaration" of procedures and functions
      used by the program and by each other
begin
    Main Program Block
end.
```

Typically the main program block is quite short and serves only to invoke some procedures or functions in the declaration portion of the program. Those procedures and functions in turn may involve others, either inside themselves or at the same level as themselves.

A major point to watch for in Pascal programs is their *hierarchical* structure as expressed by the relationships between procedures and functions.

Advantages of Procedures and Functions

Procedures and functions in Pascal may have *parameters* passed to them. This is quite unlike Integer and APPLESOFT BASIC subroutines, which are simply chunks of statements which have access to *all* the variables in the program.

The *parameters* of a procedure or function represent the information needed to carry out the purpose of that subprogram. When a procedure is invoked in a Pascal program, it is provided with the values of each parameter it needs:

```
screen(home);
sort(results,1,100);
hlin(col,col+5,row);
poke(kbdclr,0);
```

BASIC users have encountered this concept in their use of some of the *built-in* functions. They may have also defined their own parameters for user-defined functions:

```
POKE -16368,0
HLIN 10,15 AT 20
C = SCRN(ROW,COL)
X = MOD10(Y)
```

Their experience with parameters does not extend to the use of subroutines, however. BASIC subroutines are restricted to references to *global* variables, i.e., those variables whose meaning is fixed throughout an entire program. Pascal procedures do not have this limitation, and may be passed *actual* parameters which differ from invocation to invocation. The ramification of this is that in Pascal, we may write a general-purpose sorting procedure:

```
PROCEDURE sort(VAR a:values; start,stop:
INTEGER);
```

Any BASIC subroutine used for sorting is restricted to sorting one named array.

2. USER-DEFINED TYPES

In BASIC, every variable represents either an integer, a floating-point number, or a string. The only structuring mechanism available is the array. That's it! There is no other way to group logically related data together and refer to it by a single name. Therein lies the true magic and power of Pascal.

Pascal allows the user to create *names* for things—simple or complex. The beauty of that is the ability to write programs which describe a problem solution in terms of the way *we* think about that problem, *not* in terms of the way the machine views the problem.

Enumerated Types

Pascal allows us to create a list of identifiers and represent the aggregate by a single name. This has two powerful advantages:

- We can think in terms of *names*, not numbers.
- We can group logically related pieces of data together and refer to the collection by name. Variables may be created which take on values from the collection of names.

Just a few brief examples to pique your interest:

```
TYPE
  days  = (Sunday,Monday,Tuesday,
           Wednesday,Thursday,Friday,
           Saturday);
  meals = (breakfast,lunch,dinner);
  foods = (cereal,eggs,milk,cheese,bread,coffee,
           tea,beef,pork,lamb,veal,chicken,turkey,
           fish,carrots,tomatoes,peas,lettuce,
           beets,broccoli,oranges,apples,
           grapefruit,lemons,pears,peaches,
           cake,candy,cola,gingerale,icecream);
```

Records

In BASIC, variables can only hold one type of data. Pascal allows variables to contain structured data of possibly different types. Such variables are known as *records* and each component of a record is referred to as a *field*:

```
VAR
  appointment:    RECORD
                    when:      day;
                    where:     place;
                    withwhom:  person;
                  END;
```

Yes, that would be a legal Pascal statement! The even more exciting thing about records is that they may be *nested*. That means that a field of a record may itself be a record. In the example above, the type "person" could be a record:

```
person = RECORD
            name:     RECORD
                         first,
                         middle,
                         last:  string;
                      END;
            address:  RECORD
                         number: integer;
                         location: street;
                      END;
            work:     business;
         END;
```

As you can see, records may be quite complex. Yet they may be thought about in a structured and hierarchical fashion quite closely related to the external data they represent. For example, we may refer to the work of the person we wish to meet for an appointment by the Pascal expression:

```
appointment.withwhom.work
```

This would be quite different and considerably more confusing in BASIC.

Sets

Pascal allows the use of *sets*. A set variable may represent a subcollection of some enumerated type. For example:

```
TYPE
   foods = (cereal,eggs,milk,cheese,bread,coffee,
            tea,beef,pork,lamb,veal,chicken,turkey,
            fish,carrots,tomatoes,peas,lettuce,
            beets,broccoli,oranges,apples,
            grapefruit,lemons,pears,peaches,
            cake,candy,cola,gingerale,icecream);
VAR
   dairy:    set of foods;
   eaten:    foods;
   ...
   dairy := [eggs,milk,cheese];

**********************************
```

With the setup above, the statement:

```
IF eaten IN dairy
THEN
...
```

would represent a decision based on whether or not a food was eaten that happened to belong to the dairy group. Try that out in BASIC!

3. DECLARATIONS

Pascal *requires* that every variable used in a program be *declared*. There are good reasons for this. Pascal is a *strongly typed* language. That means that a *type* is assigned to every variable in a program and there is no mixing of types allowed. Since the language allows the user to define types which are not built into the language, it is necessary to explicitly tell the compiler the type of every variable, in order for it to be able to enforce the type rules of the language.

To long-time BASIC users, it may seem painful to declare all the variables of a program. However, most of these BASIC users will not make full use of the typing facilities of Pascal and hence do not see the *need* for declarations. Studying the programs of this section will provide many examples of user-defined types. We hope that if you are a long-time BASIC user who is unconvinced of the benefits of Pascal, that you will begin to see the light after reading these programs.

4. STRUCTURED STATEMENTS

Pascal offers a complete set of modern structured statements for controlling the flow of execution of a program. Integer and APPLESOFT BASIC give us IF, FOR, and GOTO. That's all! Pascal provides:

```
WHILE
IF — THEN — ELSE
FOR
REPEAT — UNTIL
CASE
GOTO
```

The other statements provide such tremendous flexibility that it is rarely necessary to use the GOTO statement.

The best way to learn the effective use of such statements is again to read real Pascal programs. The programs of this section will give you that chance. For the details of the syntax, you should consult a good introductory textbook on Pascal.

One of the most effective Pascal statements, as well as one of the most underused, is the case statement. It provides a way of selecting one of a number of statements for execution. The selection decision is made by evaluating the expression at the head of the case statement. This will result in a value which corresponds to one of the "case labels." The case labels usually come from an enumer-

ated type, which gives the case statement its unusual readability. Here's a brief example:

```
TYPE
  change = (penny,nickel,dime,quarter,half);
VAR
coin:     change;
amount:   integer;
...
  case coin of
    penny:    amount := 1;
    nickel:   amount := 5;
    dime:     amount := 10;
    quarter:  amount := 25;
    half:     amount := 50;
  end (* case coin of **);
```

This sets the variable "amount" equal to the number of cents in the kind of change represented by the value of the case selector variable, coin.

5. UCSD EXTENSIONS

APPLE Pascal is based on the UCSD Pascal system developed at the University of California at San Diego. Many features have been added to UCSD Pascal beyond those which form ISO Standard Pascal.

Strings

APPLE Pascal implements variables of type STRING. Such variables are similar to packed arrays of characters, but have an extra location as part of their storage which keeps track of the actual *length* of the string stored at any given time. This allows the length to *vary* at runtime, something which is illegal for packed arrays of characters. The length may be anywhere from 0 to 255 characters in general, although the maximum may be made shorter by explicit declaration.

Several intrinsic functions are provided to facilitate the use of strings:

i := length(s);	Provides the actual length of the string variable s. This function is quite useful in FOR statements, in determining if a string is non-empty, and so forth.

```
FOR i := 1 to length(s)
DO ...
IF length(s) > 0
THEN
    ch := s[1];
```

u := concat(s,t,...)	This function *concatenates* or sticks together two or more strings in the order named, and produces a new string as the result.
t := copy(s,start,len);	This function *extracts* a substring from one string variable and assigns it to another.

Random Access Files

APPLE Pascal provides the ability to SEEK a specific record in a file without first reading sequentially through all the records which precede it in the file. The records must be of *equal* length in order for this to work.

```
seek(f,39);
get(f);
```

would retrieve the 39th record in the file represented by file variable f. The numbering of file records begins at 0. Trying to seek a record with a negative record number produces a runtime error.

UNITS

Commonly used declarations and procedures may be placed in a "Unit," compiled by themselves, and then USEd by other programs. This reduces the amount of source that must be kept on line at compile time and allows libraries of generally useful routines to be created.

6. EXPLORATIONS

• Dig out some old BASIC programs you have written. Study them carefully and then translate them into Pascal.

Chapter

13

Pascal Trivia Quiz

In this chapter, we return once again to the APPLE Trivia Quiz program of Chapters 2 and 10. This time we redo it in Pascal, and make use of Pascal's file handling capabilities to store quizzes and other information on a diskette.

1. THE TRIVIA PROGRAM IN PASCAL

The program of Listing 13.1 presents APPLE Trivia written in Pascal. The program behaves very much like the previous versions in APPLESOFT and Integer BASIC, but it reads quite differently from those versions. We consider some of the differences in this section and the reasons for them.

Program Length

One of the first things that you notice about the Pascal version of the Trivia program is its length. It is *longer* than either BASIC version by a fair amount. There are several reasons for this, all of which are important to appreciate in order to use Pascal effectively.

- Pascal requires *all* variables to be declared.

There are lines in the var section of the program that refer not just to arrays, but to all the other variables used in the program. Pascal requires these declarations, since it checks the *type* of a variable every time it is used in a program. When two variables of different types are used in the same assignment, the compiler complains. This assists the programmer in finding many errors that would otherwise go unnoticed until runtime.

- Pascal allows const declarations and user-defined type declarations.

Strictly speaking, we could write Pascal programs which contained neither const nor type declarations. This *would* save program space—there would be fewer lines of source program—but in the long run would not be a good idea. Pascal's constants and user-defined types give the programmer significant advantages over BASIC. We shall elaborate on this thesis considerably during the course of this chapter, but let us look at one of those advantages right here.

Use constant declarations

Constant declarations make your programs:

- Easier to read and understand
- Easier to modify and extend

The APPLE Trivia Quiz declares several constants related to screen handling:

```
home   =  12;
clreol =  29;
bell   =   7;
clreos =  11;
```

Each of these represents a control character which performs some screen housekeeping function. They are used when calling the screen procedure. In the program, we see statements such as:

```
screen(home);
screen(bell);
screen(clreol);
```

as opposed to:

```
write(chr(12));
write(chr(29));
write(chr(7));
```

Now, all we *smart* programmers understand the latter form of the statements, but they are really *jargon*; slang code for the in-crowd. Beginners stumble over such statements. If it is your joy to make programs difficult to understand, then by all means use the second versions. But beware—you must read your own programs also and someday you may forget the meaning of an obscurely written statement. Enough said!

The Trivia program also contains the constant declaration:

```
responseline = 18;
```

The response line is the first of two lines on the screen where the program user keys in answers or responses to the questions in the quiz. The value of the constant *responseline* is used by different parts of the program to position the cursor and to print information on the screen near the response area.

Now suppose, for the sake of argument, that we decided to change the position of the response line on the screen: e.g., we wished to make it 17, or 19, or even 20. If we had not declared the constant *responseline*, then we would be forced to locate every occurrence of the literal value 18 in our program and change it to the new value. On the other hand, with the constant declaration, all we need to do is change the constant declaration itself. Then all references to *responseline* in the program "magically" inherit the new value so declared.

This is a general technique to use. Whenever a constant value is referred to more than once in a program, it is a good idea to declare a *name* for it in the const section. Not only does it make it easier to modify the program should that constant value need to be changed, but it also makes it easier to read the program. The name used in place of the number provides connotations and understanding that the bare number might not.

- Pascal is a free-form language

The Pascal compiler, unlike the two APPLE BASIC interpreters, does not require that a statement fit on a single line. It also does not get upset about blank lines between statements. This allows a Pascal program to spread out. The psychological advantages of the white space which this allows should not be overlooked or taken lightly.

Spread out your Pascal code

Let's look at an example. Either of the following is legal Pascal:

```
FOR i:=1 TO 10 DO BEGIN gotoxy
(20,base+i';write(letters[i],'.');END;
```

or

```
FOR i:= 1 TO 10 DO
BEGIN

  gotoxy(20,base+i);
  write(letters[i],'.');

END (* DO *);
```

We claim that the second version, although it makes the program occupy more paper, is far superior to the first.

- It is easier for the eye to follow.
- It reveals the structure of the statement better.
- It is esthetically more pleasing.

You may argue that esthetics have no place in a technical discipline. We disagree and suggest that you reconsider your position in light of the examples in this chapter.

- Pascal encourages use of procedures and functions.

Each Pascal procedure or function requires more space to write than the corresponding BASIC subroutine. This is because the Pascal procedures and functions must declare

their *parameters* in a heading. Such headings are not part of BASIC, indeed parameters to subroutines are not part of BASIC.

Each heading takes up an extra line, but communicates necessary information about how the subprogram will be used:

```
procedure screen(control:byte);
FUNCTION menu:CHAR;
```

Program Layout

In most BASIC programs, the main-line code tends to occur right at the beginning. In the Integer BASIC and APPLESOFT versions of the Trivia Quiz program, the main line consists of lines 90 to 299. Within these lines we find references to subroutines which occur later in the program.

In Pascal, the main program comes at the very end of the source. This is due to the more rigid rules of program layout in Pascal. It is not a "law of nature," simply the way the designer of the language felt it should be.

Most Pascal programs should be read from *back to front*. The Trivia program (and the others in this book) is no exception. Once you adopt the habit of starting at the back, you will find that most programs, if well-written, almost explain themselves.

Local Procedures and Functions

In BASIC, any subroutines may be GOSUBed from anywhere in the program. This is not the case in Pascal as is illustrated by the getresponses procedure. It contains within itself the code for all the following functions and procedures:

getch
answeroutline
convert
position
ahead
back
classify

Each of these performs a job that is subordinate to the work of getresponses, and is not needed anywhere else in the Trivia program. Hence, it is logical to put them *inside of* getresponses. This has several advantages:

- It prevents these procedures and functions from being used elsewhere in the program.
- It allows the program code for logically related functions to be located close together.
- It encourages top-down development of programs. In this case, getresponses could be invented early in the design process and tested in relation to other pieces of the program at its level. The capabilities internal to get responses can be added later in program development.

Sampling the keyboard in Pascal

The procedure getch used by getresponses is a tricky way to get around one of the predefined characteristics of APPLE Pascal input. With the standard function read, it is *impossible* to detect the RETURN key as input. When the user hits RETURN in response to the statement:

```
read(ch);
```

the Pascal runtime system turns the RETURN into a space! The classify procedure could not tell the difference between RETURN and SPACE using read.

We could have simply ignored this limitation and considered SPACE to be equivalent to ENTER. However, as an educational venture, we decided to provide that capability by being tricky.

The Trivia program uses a Pascal trick in order to read directly from the keyboard latch location −16384. This is accomplished in the procedure "peek" utilizing the concept of a variant record. In particular, the culprit is the user-defined record type "access." The access record type:

```
access = RECORD
            CASE BOOLEAN OF
              true: (address: INTEGER);
              false:(pointer: ↑memloc);
            END (* CASE *);
```

consists of two fields which really occupy the *same* memory space. This allows the same value to be interpreted in two different ways. The first interpretation, given by the address field of the record, considers the value stored in the variable "memory" to be the address of another APPLE RAM location. The statement:

```
memory.address:= −16384;
```

sets "memory" to be able to access location −16384. The second interpretation, given by the pointer field of the record, considers the value to be a pointer variable. The effect of the first assignment is to allow this second interpretation to determine a specific location in memory under *our* control. The statement:

```
peek := memory.pointer↑ [0];
```

uses the address to retrieve the value stored in memory location −16384. The value we get is exactly the same as if we had used the statement:

```
PEEK(-16384)
```

in BASIC.

This approach to life is not highly recommended in general. For one reason, the form of the variant record used here has been removed from standard Pascal. For another, the purpose of Pascal is to get us *away* from thinking in terms of the underlying hardware. When we lack capabilities such as the one which we just "kludged" into Pascal Trivia, we should simply adapt (e.g., consider SPACE equivalent to RETURN), or find some other way around the problem.

2. INTRODUCTION TO SEQUENTIAL FILES

In order to enhance the Pascal Trivia program, we need to review the basic concepts of Pascal sequential files.

File Variables

All files are declared in Pascal as variables:

```
file of "something"
```

where "something" stands for the *type* of the items in the file. For example:

```
VAR
  primes:   FILE OF INTEGER;
  source:   FILE OF CHAR;
  story:    FILE OF STRING;
  quizzes:  FILE OF STRING[40];
  dossier:  FILE OF personnel;
```

The "something" in each declaration is referred to as the *base* type of the file.

In APPLE Pascal, the file variable is used with the I/O intrinsics in order to **reset, rewrite, read, write, get, put,** etc. In order to use a file in a program, it must be associated with a real diskette file. This is done via the **reset** or **rewrite** intrinsics by using a string which contains the name of the desired external file. For example,

```
reset(primes,'sieve:primes.data');
reset(primes,primefn);
```

Use strings to store filenames

For maximum program flexibility, the second of the two versions of the **reset** statement is preferable to the first. It allows the actual filename string to vary from one run of the program to the next, rather than always requiring a specific file on a specific volume. The price to be paid for this flexibility is a prompt to the user for the filename:

```
write ('please enter primes filename===>');
readln(primefn);
```

File Modes

Pascal files may be open in one of two *modes*:

- Inspection
- Generation

These correspond to reading the file and writing the file, respectively. They are the only two possibilities. To open a file for inspection, use **reset.** To open a file for generation, use **rewrite.**

When a file is open for inspection, the intrinsic function *get* is used to retrieve items from that file. When a file is open for generation, the intrinsic function *put* is used to place items into that file.

Note: By definition, **rewrite(f,fn)** causes the file associated with f to become *empty*. That is, if you perform a rewrite operation on an *existing* file, all the data in that file will be *lost* when you close it. To *append* information to a file requires that a new file be created, the old file copied into it, and then data added to the new file.

The File Buffer

Associated with every Pascal file variable, f, is another variable denoted f↑, which is called the *file buffer,* or sometimes the *file window*. The file buffer is a *staging area* for elements of the file. Every piece of data coming from a file or going to a file flows through the file buffer. Here's how it works:

get(f) Places the value of the next file item into the buffer variable (unless the file is empty or the end of the file has been reached). The item may then be accessed by reference to f↑.

put(f) Places the contents of the buffer variable into the file as the next entry. This requires that a value be assigned to the buffer variable first:

```
f↑ := value;
put(f);
```

Whenever an existing file is opened with a reset, an *implicit* get is performed, placing the first item in the file into the buffer variable. This is not true of files classified by APPLE Pascal as *interactive*. Interactive files include all text files.

The program of Listing 13.2 shows a simple program which copies a file from one filename to another, removing all embedded control characters. Notice the use of the Boolean intrinsic function **eof:**

eof(f) A function which returns the value true if there are no more items to read from a given file open in inspection mode. For a file in generation mode, **eof(f)** is always true.

3. STORING QUIZZES IN FILES

In looking for ways to enhance Pascal Trivia, one thing that comes to mind is the desirability of having more quizzes. There is a limit to how many quiz procedures can be coded directly into the program. The obvious solution is to store the quizzes in files on diskette. Listing 13.3 presents the modified version of Pascal Trivia, and in the remainder of this section we discuss those changes.

The Quiz Directory

It seems simpler to group several quizzes into a single file based on broad content. Then the subject matter of all the quizzes in the file is known. This makes it easier to construct a menu of subject choices. There may be as many different subject matter files as desired, and the topic for each may be specific or fairly general.

The names of all the topics and their corresponding diskette files are stored in a *directory file*. We name the file variable which represents this file, appropriately enough, *directory*:

```
directory:    file of text;
```

The directory file contains *pairs* of strings. The first string of each pair contains a description of the topic and the second contains the filename of the diskette file containing the quizzes on that topic.

The main program of the new Pascal Trivia reads in the directory and invokes the menu procedure.

The Menu

The menu now lists the general topics for which quizzes exist. When the user chooses a topic, the choice letter is converted to a topic number and passed on to the quiz procedure.

The quiz procedure has been modified to open the appropriate quiz file, using the directory; then to read in and administer one quiz at a time until the user gets tired or until there are no more quizzes on that topic.

Limitations

The biggest limitation of this approach is that once the user leaves a topic, it is impossible to return later and pick up where he or she left off. This is due to the nature of sequential files, which must be read from start to finish, rather than sampled here and there.

4. EXPLORATIONS

- Can you figure out a way to have the procedure MARK highlight the correct answers of each quiz?
- Experiment with other kinds of quizzes such as multiple choice and true or false.
- How difficult is it to construct a program that assists in the creation of quiz files? Write such a program. Such a program should automatically update the directory file if a new category of quizzes is added. It should allow a given file of quizzes to be extended by adding new quizzes.

LISTINGS

LISTING 13.1 PASCAL TRIVIA

```
(* source file: trivia.text *)
(*$s+*)

(*****************************************)
(*                                       *)
(*  a  p  p  l  e    p  a  s  c  a  l    *)
(*                                       *)
(*                                       *)
(* ttttt rrrr  iiiii v   v iiiii aaaaa *)
(*   t   r   r   i   v   v   i   a   a *)
(*   t   r   r   i    v v    i   a   a *)
(*   t   rrrr    i    v v    i   aaaaa *)
(*   t   r r     i    v v    i   a   a *)
(*   t   r  r    i     v     i   a   a *)
(*   t   r   r iiiii    v   iiiii a   a *)
(*                                       *)
(*                                       *)
(*                                       *)
(*   by dr. richard c. vile, jr.         *)
(*   (c) 1981 - all commercial           *)
(*              rights reserved          *)
(*                                       *)
(*****************************************)

PROGRAM trivia;
USES applestuff;
CONST

  home           =       12;
  clreol         =       29;
  clreos         =       11;
  bell           =        7;

  topics         =        6;
  responseline   =       18;

  kbd            =      -16384;
  clr            =      -16368;

TYPE

  byte           =       0..255;
  memloc         =       PACKED ARRAY[0..1] OF byte;

  access  =  RECORD

               CASE BOOLEAN OF

                 true:   (address: INTEGER);
                 false:  (pointer: ^memloc);

               END (* CASE *);

VAR
```

```
  memory:         access;

  order:          ARRAY[1..10] OF INTEGER;
  useranswers:    ARRAY[1..10] OF INTEGER;

  questions:      ARRAY[1..10] OF STRING[40];
  answers:        ARRAY[1..10] OF STRING[40];
  letters:        STRING;

  base:           INTEGER;
  choice:         CHAR;
  done:           BOOLEAN;


(*****************************************)
(*          p   e   e   k                *)
(*****************************************)

FUNCTION peek(addrs: INTEGER):byte;
BEGIN

    memory.address := addrs;
    peek           := memory.pointer^[0];

END (* FUNCTION peek *);

(*****************************************)
(*          p   o   k   e                *)
(*****************************************)

PROCEDURE poke(addrs:INTEGER; val:byte);
BEGIN

    memory.address     := addrs;
    memory.pointer^[0] := val;

END (* PROCEDURE poke *);


(*****************************************)
(*          s  c  r  e  e  n             *)
(*****************************************)

PROCEDURE screen(control: byte);
BEGIN

  write(chr(control));

END (* PROCEDURE wipe *);

(*****************************************)
(*          w   a   i   t                *)
(*****************************************)
```

LISTING 13.1 (*cont.*)

```
PROCEDURE wait;
BEGIN

  gotoxy(1,23);
  write('to continue, press return...');
  readln(choice);

END (* PROCEDURE wait *);

(*****************************************)
(*  i  n  t  r  o  d  u  c  t  i  o  n *)
(*****************************************)

PROCEDURE introduction;
BEGIN

  screen(home);
  gotoxy(5,1);

  writeln('  welcome to the apple trivia quiz!');
  writeln('you will be given a choice of several');
  writeln('topics to be quizzed on.  each topic');
  writeln('will have a ten question quiz of one');
  writeln('of the following types:');
  writeln;
  writeln('      1. matching');
  writeln('      2. multiple choice');
  writeln('      3. short answer');
  writeln('      4. true or false');
  writeln;
  writeln('  you may take all the time you wish');
  writeln('to answer each quiz, but once a given');
  writeln('topic has been taken, you may not');
  writeln('return to it.');
  writeln('  good luck!!!');

  wait;

END (* PROCEDURE introduction *);

(*****************************************)
(*    i  n  i  t  i  a  l  i  z  e      *)
(*****************************************)

PROCEDURE initialize;
VAR
  i:    INTEGER;
BEGIN

  randomize;
  FOR i := 1 TO 10 DO
  BEGIN
```

```
    order[i] := i

  END (* DO *);

  letters := 'abcdefghij';

END (* PROCEDURE initialize *);

(*****************************************)
(*      a  l  p  h  a  b  e  t           *)
(*****************************************)

PROCEDURE alphabet(c:INTEGER);
VAR
  i:     INTEGER;
BEGIN

  FOR i := 1 TO 10 DO
  BEGIN

    gotoxy(c,base+i);
    write(letters[i]);
    write('.');

  END (* DO *);

END (* PROCEDURE alphabet *);

(*****************************************)
(*      s h o w q u e s t i o n s        *)
(*****************************************)

PROCEDURE showquestions;
VAR
  i:     INTEGER;
BEGIN

  FOR i := 1 TO 10 DO
  BEGIN

    gotoxy(1,base+i);
    write(questions[i]);

  END (* DO *);

END (* PROCEDURE showquestions *);

(*****************************************)
(*    s  h  o  w  a  n  s  w  e  r  s    *)
(*****************************************)

PROCEDURE showanswers(c:INTEGER);
```

LISTING 13.1 (*cont.*)

```
VAR
  i:     INTEGER;
BEGIN

  FOR i := 1 TO 10 DO
  BEGIN

    gotoxy(c,base+i);
    write(answers[order[i]]);

  END (* DO *);

END (* PROCEDURE showanswers *);

(*****************************************)
(*        m   i   x   u   p              *)
(*****************************************)

PROCEDURE mixup;
VAR
  i,j:     INTEGER;
BEGIN

  FOR i := 1 TO 10 DO
    order[i] := -1
  (* enddo *);

  FOR i := 1 TO 10 DO
  BEGIN

    REPEAT
      j := (1 + random MOD 10);
    UNTIL order[j] = -1;
    order[j] := i;

  END (* DO *);

END (* PROCEDURE mixup *);

(*****************************************)
(*  g  e  t  r  e  s  p  o  n  s  e  s *)
(*****************************************)

PROCEDURE getresponses;
TYPE
  reply = (quizlet,backarrow,rarrow,return,none);
VAR
  i,row,col,spot:        INTEGER;
  done:                  BOOLEAN;
  ch:                    CHAR;

  (*=====================================*)
  PROCEDURE getch(VAR c:CHAR);
```

```
BEGIN

  REPEAT
  UNTIL peek(kbd)>128;

  c := chr(peek(kbd)-128);
  poke(clr,0);

END (* PROCEDURE getch *);

(*=======================================*)
PROCEDURE answeroutline;
BEGIN

  gotoxy(1,responseline - 2);
  writeln('*****************************************');
  writeln;
  writeln('1.      2.      3.      4.      5.');
  writeln('6.      7.      8.      9.     10.');
  writeln;
  writeln('move the cursor with right and left');
  writeln('arrows. ===>hit escape to finish<===');

END (* PROCEDURE answeroutline *);

(*=======================================*)
PROCEDURE convert;
BEGIN

  row := spot DIV 6;
  col := (spot-1)MOD 5;

END (* PROCEDURE convert *);

(*=======================================*)
PROCEDURE position;
BEGIN

  convert;
  gotoxy(7*col+3,responseline+row);

END (* PROCEDURE position *);

(*=======================================*)
PROCEDURE ahead;
BEGIN

  spot := spot + 1;
  IF spot = 11
  THEN
    spot := 1
  (* endif *);
```

LISTING 13.1 *(cont.)*

```
END (* PROCEDURE ahead *);

(*=====================================*)
PROCEDURE back;
BEGIN

  spot := spot - 1;
  IF spot = 0
  THEN
    spot := 10
  (* endif *);

END (* PROCEDURE back *);

(*=====================================*)
FUNCTION classify:reply;
CONST
  orda   =  65;
  ordz   =  90;
  ordback = 8;
  ordforw = 21;
  ordcr   = 13;

VAR
  chvalue: INTEGER;

BEGIN

  getch(ch);
  chvalue := ord(ch);

  IF (chvalue>=orda) AND (chvalue<=ordz)
  THEN
    classify := quizlet
  ELSE
    IF chvalue = ordback
    THEN
      classify := backarrow
    ELSE
      IF chvalue = ordforw
      THEN
        classify := rarrow
      ELSE
        IF chvalue = ordcr
        THEN
          classify := return
        ELSE
          classify := none
        (* END IF chvalue = ordcr *)
      (* END IF chvalue = ordforw *)
    (* END IF chvalue = ordback *)
  (* END IF (chvalue>=orda... *);
```

```
  END (* FUNCTION classify *);

BEGIN (* ======> getresponses <====== *)

  spot := 1;
  answeroutline;

  FOR i := 1 TO 10 DO
    useranswers[i] := 0
  (* enddo *);

  done := false;

  REPEAT

    position;

    CASE classify OF

      quizlet: BEGIN
                 write(ch);
                 useranswers[spot] := (ord(ch)-ord('a')+1);
                 ahead;
               END;

      backarrow:  back;
      rarrow:     ahead;
      return:     done := true;
      none:       ;

    END (* CASE classify OF *);

  UNTIL done;

END (* PROCEDURE getresponses *);

(*****************************************)
(*          m    a    r    k             *)
(*****************************************)

PROCEDURE mark;
VAR
  right,i:       INTEGER;
BEGIN

  right := 0;
  FOR i := 1 TO 10 DO
  BEGIN

    IF useranswers[order[i]] = i
    THEN
      right := right + 1
    (* endif *);
```

LISTING 13.1 *(cont.)*

```
  END (* DO *);

  gotoxy(1,20);
  writeln;
  writeln;
  screen(clreol);
  write('you got ',right:3,' correct');

END (* PROCEDURE mark *);

(*****************************************)
(*          q   u   i   z                *)
(*****************************************)

PROCEDURE quiz(c:INTEGER);
BEGIN

  showquestions;
  alphabet(c);
  mixup;
  showanswers(c+3);
  getresponses;
  mark;
  wait;

END (* PROCEDURE quiz *);

(*****************************************)
(*      t   o   l   k   i   e   n        *)
(*****************************************)

PROCEDURE tolkien;
BEGIN

  screen(home);
  writeln;
  writeln('match the names of the following world');
  writeln('tolkien characters with information');
  writeln('about them:');
  writeln;

  questions[1]  := '1. landroval';
  questions[2]  := '2. frodo';
  questions[3]  := '3. gollum';
  questions[4]  := '4. dunadan';
  questions[5]  := '5. bill ferny';
  questions[6]  := '6. ugluk';
  questions[7]  := '7. lotho';
  questions[8]  := '8. mumak';
  questions[9]  := '9. grima';
  questions[10] := '10.fimbrethil';
```

```
  answers[1]  := 'gwaihir''s brother';
  answers[2]  := 'wore the mithril';
  answers[3]  := 'friend of deagol';
  answers[4]  := 'a ranger';
  answers[5]  := 'sold sam his pony';
  answers[6]  := 'captain of uruk hai';
  answers[7]  := 'a sackville-baggins';
  answers[8]  := 'an oliphaunt';
  answers[9]  := 'wormtongue';
  answers[10] := 'one of the entwives';


  base := 4;
  quiz(17);

END (* PROCEDURE tolkien *);

(*****************************************)
(*     c  a  p  i  t  a  l  s            *)
(*****************************************)

PROCEDURE capitals;
BEGIN

  screen(home);
  writeln;
  writeln('match the following countries to');
  writeln('their capital cities:');
  writeln;

  questions[1]  := '1. rumania';
  questions[2]  := '2. yugoslavia';
  questions[3]  := '3. turkey';
  questions[4]  := '4. chile';
  questions[5]  := '5. columbia';
  questions[6]  := '6. haiti';
  questions[7]  := '7. jordan';
  questions[8]  := '8. rhodesia';
  questions[9]  := '9. saudia arabia';
  questions[10] := '10.mongolia';

  answers[1]  := 'bucharest';
  answers[2]  := 'sofia';
  answers[3]  := 'ankara';
  answers[4]  := 'santiago';
  answers[5]  := 'la paz';
  answers[6]  := 'port au prince';
  answers[7]  := 'amman';
  answers[8]  := 'salisbury';
  answers[9]  := 'riyad';
  answers[10] := 'ulan bator';

  base := 3;
```

LISTING 13.1 (*cont.*)

```
  quiz(19);

END (* PROCEDURE capitals *);

(*****************************************)
(*    b o o k s a n d a u t h o r s      *)
(*****************************************)

PROCEDURE booksandauthors;
BEGIN

END (* PROCEDURE booksandauthors *);

(*****************************************)
(*       c o m p u t e r l o r e         *)
(*****************************************)

PROCEDURE computerlore;
BEGIN

END (* PROCEDURE computerlore *);

(*****************************************)
(* g e n e r a l i n f o r m a t i o n *)
(*****************************************)

PROCEDURE generalinformation;
BEGIN

END (* PROCEDURE generalinformation *);

(*****************************************)
(*            m   e   n   u              *)
(*****************************************)

FUNCTION menu:CHAR;
BEGIN

  screen(home);
  gotoxy(5,5);

  writeln('choose one of the following:');
  writeln;
  writeln('      [a]  tolkien');
  writeln('      [b]  capitals');
  writeln('      [c]  books and authors');
  writeln('      [d]  computer lore');
  writeln('      [e]  general information');
  writeln('      [f]  exit');

  REPEAT

    screen(bell);
```

```
    gotoxy(5,14);
    screen(clreol);
    read(choice);

  UNTIL ( choice >= 'a')
        AND
        ( choice <= 'f');

  menu := choice;

END (* FUNCTION menu *);




          (*************************************************)
          (*                                               *)
BEGIN     (* ==========>  t  r  i  v  i  a  <============ *)
          (*                                               *)
          (*************************************************)

  introduction;
  initialize;

  done := false;
  REPEAT

    CASE menu OF

      'a':      tolkien;
      'b':      capitals;
      'c':      booksandauthors;
      'd':      computerlore;
      'e':      generalinfo;
      'f':      done := true;

    END (* CASE *)

  UNTIL done;

END (* PROGRAM trivia *).
```

LISTING 13.2 COPY PROGRAM DEMO

```
(*************************************)
(*                                   *)
(* this program copies one sequential *)
(* file to another and removes the    *)
(* control characters embedded in the *)
(* first file in the process.         *)
(*                                   *)
(*************************************)

PROGRAM copynocontrol;

TYPE

  controlchar    =      0..26;

VAR

  source:         text;
  dest:           text;
  sourcename:     STRING;
  destname:       STRING;

BEGIN

  write('source file?===>');
  readln(sourcename);

  write('destination file?===>');
  readln(destname);

  reset(source,sourcename);
  rewrite(dest,destname);

  REPEAT

    get(source);
    IF ord(source^) >= 27
    THEN
    BEGIN

      dest^ := source^;
      put(dest);

    END (* IF ord(source^) >= 27 *);

  UNTIL eof(source);

  close(dest,lock);
  close(source,lock);

END.
```

LISTING 13.3 APPLE TRIVIA WITH FILES

```
(* source file: ftriv.text *)
(*$s+*)

(*****************************************)
(*                                       *)
(*  a  p  p  l  e    p  a  s  c  a  l   *)
(*                                       *)
(*                                       *)
(* ttttt rrrr  iiiii v   v iiiii aaaaa *)
(*   t   r   r   i   v   v   i   a   a *)
(*   t   r   r   i    v v    i   a   a *)
(*   t   rrrr    i    v v    i   aaaaa *)
(*   t   r r     i    v v    i   a   a *)
(*   t   r  r    i     v     i   a   a *)
(*   t   r   r iiiii   v   iiiii a   a *)
(*                                       *)
(*                                       *)
(*                                       *)
(*   by dr. richard c. vile, jr.         *)
(*   (c) 1981 - all commercial           *)
(*              rights reserved          *)
(*                                       *)
(*****************************************)

PROGRAM trivia;
USES applestuff;
CONST

  home           =       12;
  clreol         =       29;
  clreos         =       11;
  bell           =        7;

  topics         =        6;
  responseline   =       18;

  kbd            =   -16384;
  clr            =   -16368;

TYPE

  byte           =       0..255;
  memloc         =       PACKED ARRAY[0..1] OF byte;

  access  =  RECORD

               CASE BOOLEAN OF

                 true:   (address: INTEGER);
                 false:  (pointer: ^memloc);

               END (* CASE *);

VAR
```

LISTING 13.3 *(cont.)*

```
  directory:      text;
  quizzes:        text;

  dir:            ARRAY[1..20] OF STRING[40];
  dirfn:          ARRAY[1..20] OF STRING[40];

  i:              INTEGER;
  numtopics:      INTEGER;

  memory:         access;

  order:          ARRAY[1..10] OF INTEGER;
  useranswers:    ARRAY[1..10] OF INTEGER;

  questions:      ARRAY[1..10] OF STRING[40];
  answers:        ARRAY[1..10] OF STRING[40];
  letters:        STRING;
  line:           STRING;

  row:            INTEGER;
  col:            INTEGER;
  spot:           INTEGER;
  base:           INTEGER;
  choice:         CHAR;
  done:           BOOLEAN;


(*****************************************)
(*          p   e   e   k                *)
(*****************************************)

FUNCTION peek(addrs: INTEGER):byte;
BEGIN

    memory.address := addrs;
    peek           := memory.pointer^[0];

END (* FUNCTION peek *);

(*****************************************)
(*          p   o   k   e                *)
(*****************************************)

PROCEDURE poke(addrs:INTEGER; val:byte);
BEGIN

    memory.address     := addrs;
    memory.pointer^[0] := val;

END (* PROCEDURE poke *);
```

```
(*****************************************)
(*          s  c  r  e  e  n             *)
(*****************************************)

PROCEDURE screen(control: byte);
BEGIN

  write(chr(control));

END (* PROCEDURE wipe *);

(*****************************************)
(*           w   a   i   t               *)
(*****************************************)

FUNCTION wait:BOOLEAN;
BEGIN

  gotoxy(1,23);
  write('another quiz on the same topic(y/n)?');
  readln(choice);
  IF (choice = 'y')
  THEN
    wait := true
  ELSE
    wait := false
  (* endif *);

END (* PROCEDURE wait *);

(*****************************************)
(*  i  n  t  r  o  d  u  c  t  i  o  n *)
(*****************************************)

PROCEDURE introduction;
BEGIN

  screen(home);
  gotoxy(1,1);

  writeln('  welcome to the apple trivia quiz!');
  writeln('you will be given a choice of several');
  writeln('topics to be quizzed on.  each topic');
  writeln('will have a ten question quiz of one');
  writeln('of the following types:');
  writeln;
  writeln('      1. matching');
  writeln('      2. multiple choice');
  writeln('      3. short answer');
  writeln('      4. true or false');
  writeln;
  writeln('  you may take all the time you wish');
  writeln('to answer each quiz, but once a given');
```

LISTING 13.3 (*cont.*)

```
  writeln('topic has been taken, you may not');
  writeln('return to it.');
  writeln('  good luck!!!');

  gotoxy(1,23);
  write('to continue, press return...');
  readln(choice);
  screen(home);

END (* PROCEDURE introduction *);

(*****************************************)
(*      s h o w q u e s t i o n s        *)
(*****************************************)

PROCEDURE showquestions;
VAR
  i:     INTEGER;
BEGIN

  FOR i := 1 TO 10 DO
  BEGIN

    gotoxy(0,base+i);
    write(i:2,'. ',questions[i]);

  END (* DO *);

END (* PROCEDURE showquestions *);

(*****************************************)
(*    s  h  o  w  a  n  s  w  e  r  s    *)
(*****************************************)

PROCEDURE showanswers(c:INTEGER);
VAR
  i:     INTEGER;

  (*================================*)
  PROCEDURE mixup;
  VAR
    i,j:     INTEGER;
  BEGIN

    randomize;

    FOR i := 1 TO 10 DO
      order[i] := -1
    (* enddo *);

    FOR i := 1 TO 10 DO
    BEGIN
```

```
      REPEAT
        j := (1 + random MOD 10);
      UNTIL order[j] = -1;
      order[j] := i;

    END (* DO *);

  END (* PROCEDURE mixup *);

  (*==============================*)
  PROCEDURE alphabet(c:INTEGER);
  VAR
    i:     INTEGER;
  BEGIN

    letters := 'abcdefghij';

    FOR i := 1 TO 10 DO
    BEGIN

      gotoxy(c,base+i);
      write(letters[i]);
      write('.');

    END (* DO *);

  END (* PROCEDURE alphabet *);

BEGIN (* =====> showanswers <===== *)

  mixup;
  alphabet(c-3);

  FOR i := 1 TO 10 DO
  BEGIN

    gotoxy(c,base+i);
    write(answers[order[i]]);

  END (* DO *);

END (* PROCEDURE showanswers *);

(*****************************************)
(*    p   o   s   i   t   i   o   n      *)
(*****************************************)

PROCEDURE position;

  (*==============================*)
  PROCEDURE convert;
  BEGIN
```

LISTING 13.3 *(cont.)*

```
    row := spot DIV 6;
    col := (spot-1)MOD 5;

  END (* PROCEDURE convert *);

BEGIN  (* =====> position <===== *)

  convert;
  gotoxy(7*col+3,responseline+row);

END (* PROCEDURE position *);

(*****************************************)
(*  g  e  t  r  e  s  p  o  n  s  e  s *)
(*****************************************)

PROCEDURE getresponses;
TYPE
  reply = (quizlet,backarrow,rarrow,return,none);
VAR
  i:                    INTEGER;
  done:                 BOOLEAN;
  ch:                   CHAR;

  (*=====================================*)
  PROCEDURE getch(VAR c:CHAR);
  BEGIN

    REPEAT
    UNTIL peek(kbd)>128;

    c := chr(peek(kbd)-128);
    poke(clr,0);

  END (* PROCEDURE getch *);

  (*=====================================*)
  PROCEDURE answeroutline;
  BEGIN

    gotoxy(1,responseline - 2);
    writeln('*****************************************');
    writeln;
    writeln('1.     2.     3.     4.     5.');
    writeln('6.     7.     8.     9.    10.');
    writeln;
    writeln('move the cursor with right and left');
    writeln('arrows. ===>hit return to finish<===');

  END (* PROCEDURE answeroutline *);

  (*=====================================*)
  PROCEDURE ahead;
```

```
BEGIN

  spot := spot + 1;
  IF spot = 11
  THEN
    spot := 1
  (* endif *);

END (* PROCEDURE ahead *);

(*=====================================*)
PROCEDURE back;
BEGIN

  spot := spot - 1;
  IF spot = 0
  THEN
    spot := 10
  (* endif *);

END (* PROCEDURE back *);

(*=====================================*)
FUNCTION classify:reply;
CONST
  orda   =  65;
  ordz   =  90;
  ordback = 8;
  ordforw = 21;
  ordcr   = 13;

VAR
  chvalue: INTEGER;

BEGIN

  getch(ch);
  chvalue := ord(ch);

  IF (chvalue>=orda) AND (chvalue<=ordz)
  THEN
    classify := quizlet
  ELSE
    IF chvalue = ordback
    THEN
      classify := backarrow
    ELSE
      IF chvalue = ordforw
      THEN
        classify := rarrow
      ELSE
        IF chvalue = ordcr
        THEN
```

LISTING 13.3 (*cont.*)

```
            classify := return
          ELSE
            classify := none
          (* END IF chvalue = ordcr *)
        (* END IF chvalue = ordforw *)
      (* END IF chvalue = ordback *)
    (* END IF (chvalue>=orda... *);

  END (* FUNCTION classify *);

BEGIN (* =====> getresponses <===== *)

  spot := 1;
  answeroutline;

  FOR i := 1 TO 10 DO
    useranswers[i] := 0
  (* enddo *);

  done := false;

  REPEAT

    position;

    CASE classify OF

      quizlet: BEGIN
                 write(ch);
                 useranswers[spot] := (ord(ch)-ord('a')+1);
                 ahead;
               END;

      backarrow:  back;
      rarrow:     ahead;
      return:     done := true;
      none:       ;

    END (* CASE classify OF *);

  UNTIL done;

END (* PROCEDURE getresponses *);

(*****************************************)
(*          m   a   r   k                *)
(*****************************************)

PROCEDURE mark;
VAR
  right,i:       INTEGER;
BEGIN
```

```
  right := 0;
  FOR i := 1 TO 10 DO
  BEGIN

    IF useranswers[order[i]] = i
    THEN
      right := right + 1
    ELSE
    BEGIN
      spot := order[i];
      position;
      write('x');
    END (* IF *);

  END (* DO *);

  gotoxy(0,21);
  screen(clreos);
  FOR i := 1 TO right DO
    screen(bell);
  writeln('you got ',right:3,' correct');
  writeln('x''s show incorrect answers');

END (* PROCEDURE mark *);

(*****************************************)
(*      g   e   t   q   u   i   z        *)
(*****************************************)

PROCEDURE getquiz;
VAR
  i:     INTEGER;

BEGIN (* getquiz *)


  FOR i := 1 TO 10 DO
  BEGIN

    readln(quizzes,questions[i]);

  END (* DO *);

  FOR i := 1 TO 10 DO
  BEGIN

    readln(quizzes,answers[i]);

  END (* DO *);

END (* PROCEDURE getquiz *);
```

LISTING 13.3 (*cont.*)

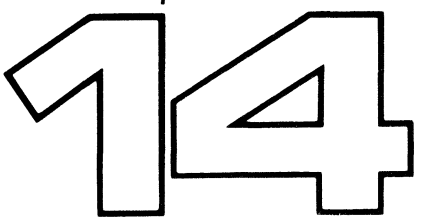
```
(*****************************************)
(*          q    u    i    z               *)
(*****************************************)

PROCEDURE quiz(which:INTEGER);
VAR
  c:  INTEGER;

  (*==============================*)
  FUNCTION convert(v:STRING):INTEGER;
  VAR
    value,i: INTEGER;
  BEGIN

    value := 0;
    FOR i := 1 TO length(v) DO
      value := value*10+(ord(v[i])-ord('0'));

    convert := value;

  END (* FUNCTION convert *);

BEGIN (* =====> quiz <===== *)

  reset(quizzes,dirfn[which]);
  done := false;

  WHILE NOT done DO
  BEGIN


    screen(home);

    (* determine the offset on the screen    *)
    (* at which the answers will be printed *)

    readln(quizzes,line);
    c := convert(line);

    gotoxy(1,0);
    writeln('topic: ',dir[which]);

    writeln;
    base := 1;
    repeat

      readln(quizzes,line);
      writeln(line);
      base := base + 1;

    UNTIL line[length(line)] = ':';

    getquiz;
```

```
      showquestions;
      showanswers(c+3);
      getresponses;
      mark;
      done := NOT wait;   (* wait returns true IF the *)
                          (* user wants more.         *)

      IF (NOT done) AND (eof(quizzes))
      THEN
      BEGIN

        done := true;
        writeln('there are no more quizzes on ',dir[which]);
        write('to continue, press return...');
        readln(choice);

      END (* IF NOT done AND eof *);

    END (* WHILE NOT done DO *);

  END (* PROCEDURE quiz *);

(*****************************************)
(*          m   e   n   u                *)
(*****************************************)

PROCEDURE menu;
VAR
  done: BOOLEAN;
BEGIN

  done := false;

  REPEAT

    screen(home);
    gotoxy(1,1);

    writeln('quizzes are available on the');
    writeln('following topics.  choose one:');
    writeln;

    for i := 1 to numtopics do
    begin

      write('     [',chr(ord('')+i),']  ');
      writeln(dir[i]);

    END (* DO *);

    REPEAT

      screen(bell);
```

LISTING 13.3 (*cont.*)

```
      gotoxy(5,14);
      screen(clreol);
      read(choice);

    UNTIL ( choice >= 'a')
          AND
          ( choice <= chr(ord('')+numtopics));

    quiz(ord(choice) - ord(''));

    screen(home);
    gotoxy(5,5);
    write('another topic?(y/n)===>');
    readln(choice);
    IF choice <> 'y'
    THEN
      done := true
    (* endif *);
    close(quizzes, lock);

  UNTIL done;

END (* PROCEDURE menu *);

            (***********************************************)
            (*                                             *)
BEGIN       (* =========>  t  r  i  v  i  a  <========== *)
            (*                                             *)
            (***********************************************)

  introduction;
  reset(directory,'qdir.text');

  readln(directory,dir[1]);
  readln(directory,dirfn[1]);

  i := 2;

  WHILE NOT eof(directory) DO
  BEGIN

    readln(directory,dir[i]);
    readln(directory,dirfn[i]);
    i := i + 1;

  END (* WHILE NOT eof *);

  numtopics := i - 1;
  menu;
  screen(home);   (* clean up the screen *)

END (* PROGRAM tRIVIA *).
```

Chapter



Interactive Programs: Adding Intelligence

Many interactive programs have too many options to rely solely on menus for user interaction. Programs such as command languages, interpreters for BASIC, fancy adventure games, and even certain complex application programs may accept long strings of user commands, which must be broken down and analyzed in order for the program to act intelligently. In this chapter we shall discuss the necessary Pascal tools for breaking down strings of characters into meaningful pieces. The process by which this is accomplished is known as *scanning*, or *lexical analysis*. At the end of the chapter a collection of Pascal procedures is presented which performs lexical analysis. They may be used as a skeleton upon which to build many useful applications.

1. MORE FLEXIBLE USER INPUT

In order to be able to analyze a complex command, it must first be broken down into its component parts. Those parts are usually referred to in the language of computer science as *tokens*. If necessary, some of them must be converted into a different form, more convenient for the computer to manipulate.

Tokens

In many cities with subways, tokens are a form of currency; your ticket to ride, so to speak. In computer lingo, tokens are recognizable chunks of an input stream. Tokens can be either single characters or strings of characters—depending mostly on the function served by the token.

In programming languages, tokens typically are such things as:

Identifiers
Numbers
Labels (or line numbers)
Operators
  single character: +,−, *,/,=,<>
  multiple character:
  AND,OR,NOT,<=,>=,<>
Delimiters: single characters which serve to mark the beginning or the end of some construct: e.g., (,) which delimit an expression.

There are many variations from language to language, but these examples should give you the basic idea.

In other applications, tokens may be simpler in structure. For example, in a word processor a token might be a

nonblank sequence of characters. This would be called a "word" for simplicity, even though some "words" might not be those we use in everyday speech.

In Chapter 9, we presented two subroutines in APPLESOFT, each designed to split up a string of characters into its constituent *words* (according to the above definition). The procedure of Listing 14.1 is a Pascal procedure which performs the same task.

2. THE STRUCTURE OF SCANNING PROGRAMS

A typical scanning procedure needs a great deal of help in carrying out its tasks. This help usually takes the form of subsidiary procedures which interface to the outside world or perform certain common steps necessary in the scanning process. Before going into the details of the token extraction process, we first discuss the overall structure of the scanner and its satellite procedures.

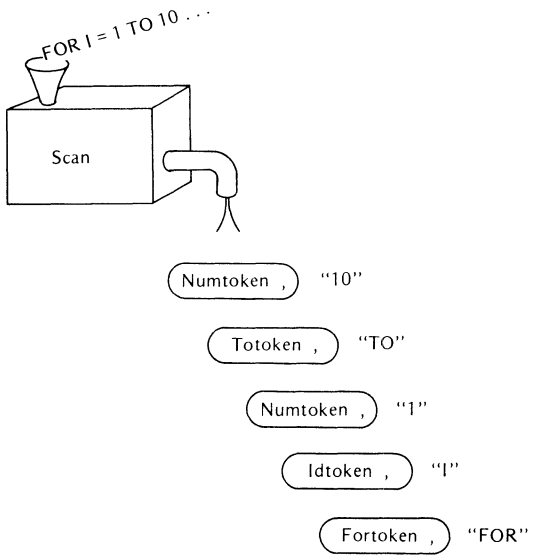
Figure 14.1 illustrates the functional concept of a scanner: In goes a stream of characters and out comes a sequence of tokens. The figure shows tokens as pairs. This implies that the scanner does two things:

1. It collects the characters of the next token into one place, usually a string variable, or an array of characters. The variable or array used as the collection "basket" is commonly called the token buffer.
2. It returns a value that represents which token was extracted. This value is an internal "name" for the token. In Pascal this may be conveniently represented as a value from an enumerated type:

```
tokenvalue = (idtoken,numtoken,equalstoken,
              plustoken,fortoken,totoken,...);
```

The internal name for the token may be used by the caller of the scanner to perform further analysis. Such analysis might involve translating a command into a different form, such as machine language; or perhaps interpreting a statement in BASIC. We shall see examples of this in Chapter 15. For now, however, we shall simply be interested in how the scanner determines which token value to return.

FIGURE 14.1 Input and Output of Scan



Scan and Its Subordinates

Figure 14.2 shows the support procedures needed by the main Scan procedure. We shall explain each below. First let us look at the insides of Scan itself.

Scan is a function which "computes" the value of the next token to be found in an input character string. The character string may be provided to Scan as an actual parameter when Scan is invoked, or it may be a global variable of the program which uses Scan. In either case, a mechanism must be provided which keeps track of where Scan should pick up or continue in the input string.

The major part of the Scan function is a case statement which is controlled by an expression which classifies the next character of the input. Each individual case is responsible for recognizing the possible tokens which may begin with a particular class of characters. For example, if the first character of the token is a letter, then the token itself might be a user variable name or a reserved-word identifier. The code inside each case is specially designed to recognize specific tokens and makes use of some of the support routines indicated in Figure 14.2.

Character Classes

The case statement in Scan assumes two things:

1. A categorization has been decided upon which places each possible input character into a class.
2. A routine is available to decide for each input character which class it belongs to.

Certain natural classes of characters spring to mind almost immediately: namely, digits and letters.

Digits	if we wish to extract numbers, in particular integers, it is a must to be able to recognize digits.
Letters	letters are the principal ingredients of names, keywords, identifiers, etc.

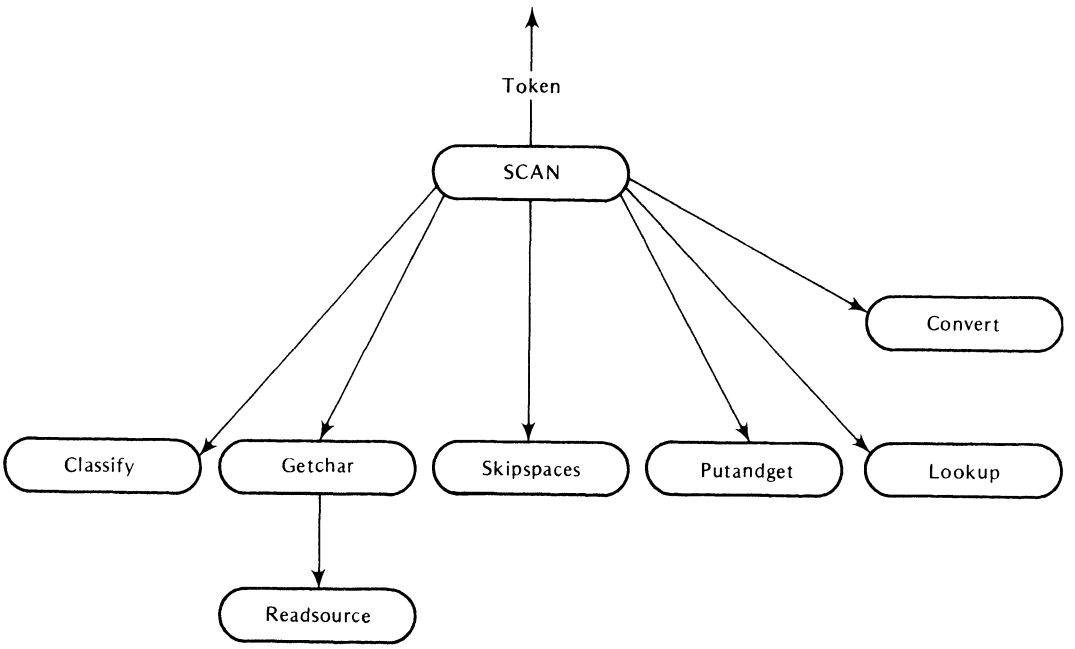


FIGURE 14.2 Scan and Its Subordinates

In almost every scanning application you will ever encounter, the character classes of *digit* and *letter* will take part, and be interpreted in the same way—namely, the everyday interpretation of these terms.

All other characters that a user might type in a command or use in a program could be lumped into a single class. It could be called the "others" class for want of a better term. Usually, however, the other characters are themselves divided up into one or more categories depending on the nature of the legal commands or programs being scanned. For example, a BASIC interpreter might classify +,−,/,*,<,>,= as OPERATOR characters, whereas an adventure game would probably consider them as "noise" or insignificant characters.

Commonly used terms for certain classes of "other" characters are:

Operators
Delimiters
Special characters

The term *delimiter* usually refers to a character such as ",", ".", ";", "(", and ")" which serves to delimit or separate one part of a command from another. For example, commas serve to delimit the individual variables in a list of variables:

```
INPUT A,B,C
```

In many contexts, certain characters may be in a class all by themselves. This is not an absolute law of nature, but rather a practical decision based on the details of writing a scanning program. For example, many expression processing programs will consider left and right parentheses to be character classes, each consisting of a single character. Recognition of character classes plays an important role in the overall process of token extraction as we shall now see.

Classify

Classify is itself a function subprogram. It takes a single character as input and returns a value from an enumerated type which represents the possible character classes for the particular scanner in question.

The classify function is not always used. In many scanners its purpose is absorbed into the main case statement of Scan itself. We present an example of the latter approach in Chapter 15.

Handing Characters to Scan

The scanner calls upon a support routine each time it requires a new character for analysis. This allows the handling of input to be separated from the analysis of that input.

Getchar

Getchar is a function which returns the next character of the input when called:

```
ch := getchar;
```

Getchar plucks individual characters from some source, such as a string variable, a packed array of characters, or simply the next character returned by a call of the Pascal **get** function.

Getchar plays a very important role in the design of a scanning program. It stands *between* the rest of the scanner and the detailed part of the program responsible for I/O. This means that you may write a large part of the code for the scanner without worrying about such details as what happens when the current string or array runs out of characters. Getchar will see to it that a character is always returned, regardless of how input is being done.

In most scanners, getchar is responsible for filling up the input buffer whenever it becomes empty. This is usually done by calling another procedure, readsource, which is responsible for obtaining more input. Readsource may input a line of text from a source file or it may prompt for and obtain a line of text from an interactive terminal. In any case, it returns a line of text to getchar until it runs out of input. This may occur, for example, by reaching end of file.

When getchar is no longer able to get characters from readsource, it will signal by returning some special character which is unused by the scanner for any other purpose. In the skeleton scanner presented later, the character % is used for this purpose. This tells the scanner that no more characters are available.

Other Support Routines

In addition to classify and getchar, the scanner will use the following routines:

Skipspaces

This procedure is called whenever it is necessary to skip over one or more spaces in the input. Upon return from this procedure, it is guaranteed that the variable nextchar will contain the next nonblank character from the input. In particular, it is safe to call skipspaces any time: if the current character is already nonblank, the result will be to do nothing.

Putandget

This procedure causes the most recently retrieved character to be added to the token buffer. After doing that the procedure will call getchar to set nextchar to a new value.

Putandget may be thought of as the bucket brigade of the scanner. It "puts" a character which it already has in hand and "gets" the next.

Convert

This procedure converts a string of digits representing an integer into the corresponding numeric value to be stored in an integer variable. This value may then be used during further analysis.

Lookup

In case there is a need to distinguish certain words that have a predefined meaning from other words which might be used as identifiers, this procedure will make a search for such words. Words with predefined meanings are called *reserved words*. In Pascal words such as IF, CASE, PROCEDURE, PROGRAM, etc. are reserved words. They may not be used as identifiers in a program.

3. SOME EXAMPLES OF TOKEN EXTRACTION

As mentioned above, a typical approach to token recognition begins by classifying the first character of each token. The class of the first character narrows the possible tokens to one or two in most instances. Once the initial character class is recognized, special processing for the expected token(s) may then be performed. We now consider some examples of such special processing.

Extracting Integers

An integer has the structure of a sequence of digits. Once the scanner recognizes the first digit of the sequence, the rule for extracting the whole integer is simple: keep picking off the next character until one is found which is *not* a digit. Here we can see the utility of the putandget routine.

Since putandget puts first and gets second, the scanner can always know what class of character is being stored into the token buffer and may always examine the next character before deciding what to do with it.

This really isn't the whole story, since it may be necessary to convert the number into an integer in numeric rather than string from.

Using set variables in scanning

In order to turn the process just discussed into Pascal code, we might imagine the following:

```
done := false;
REPEAT
  IF (nextchar = '0') OR
  (nextchar = '1') OR
  (nextchar = '2') OR
  (nextchar = '3') OR
  (nextchar = '4') OR
  (nextchar = '5') OR
  (nextchar = '6') OR
  (nextchar = '7') OR
  (nextchar = '8') OR
  (nextchar = '9')
  THEN
    putandget
  ELSE
    done := true
    (* endif *);
UNTIL done;
```

This code gets the job done, but it is cumbersome. It may be considerably simplified by the use of a set variable. Suppose we declare:

```
var
  digits:  set of char;
```

and then assign:

```
digits := ['0'..'9'];
```

Then the variable **digits** represent a set of characters which contains precisely the decimal digits. Then the Pascal expression:

```
nextchar (in digits)
```

will be true if and only if the character stored in **nextchar** is in fact a decimal digit. This leads to the following rewrite of the above statements:

```
WHILE (nextchar IN digits) DO
  putandget;
```

This does in two lines what we took 18 lines to do above! Not only that but it is much clearer and easier to understand.

Moral: Never measure a programmer's worth solely in number of lines of code written per day! It's not only quantity that counts but quality as well.

The use of variables of type "set of char" is typical of scanning programs written in Pascal.

Extracting Identifiers

By now you begin to see how easy it is to write scanning code in Pascal. Now think about identifiers. An identifier is a string of characters which begins with a letter and is followed by letters or digits or in some cases certain special characters such as "$", ".", or "_" (underscore). The permissible length of an identifier, the special characters allowed in an identifier, and any other restrictions or permissions having to do with an identifier depend on the context in which the identifier appears.

Looking back at the Pascal statements for extracting a number, we can easily come up with the following:

```
WHILE (nextchar IN letters) OR
      (nextchar IN digits)
DO
  putandget;
```

This statement would be executed only after first having scanned the initial letter of the identifier.

4. HANDLING RESERVED WORDS

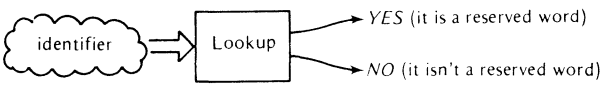
A *reserved word* is not an introverted part of speech. It is a token which has the structure of an identifier but a special meaning all its own—reserved for it if you will. When a scanner extracts an identifier, it may have the additional duty of deciding whether the identifier is a reserved word or not. In order to accomplish this task, it will use the lookup routine mentioned earlier.

Lookup is a function which takes an identifier as input and returns a Boolean result. The Boolean return value indicates whether or not the identifier matched one of the reserved words. This is illustrated in Figure 14.3. The reserved words are stored as an array of strings:

```
VAR
  rw: ARRAY[1..maxrw] of STRING[40];
```

The size limitation is optional, of course, but 40 seems like a reasonable upper bound for the length of a reserved word. The internal algorithm used by lookup is a linear

FIGURE 14.3 The Lookup Function and Its Results



search: that is, a single pass through the reserved words array in sequential order.

The linear search algorithm is a common one, used daily in numerous programming applications. In implementing it, the usual technique is to use a for loop:

```
FOR i := 1 TO maxrw do
```

Check the ith reserved word for equality with the search string. Of course, if the test for equality succeeds, it is unnecessary to proceed through the remainder of the array. Consequently, provision must be made for a way to escape from the loop. This usually involves a goto statement and an extra test outside the loop:

```
FOR i := 1 TO maxrw do
  IF search = rw[i]
  THEN
    goto 99;
99: IF i <= maxrw THEN found := true;
```

This code is relatively simple and easy to understand, but it involves unnecessary complexity. Each time through the loop there are two tests: the if test inside the loop and the *implicit* test of the for statement itself, checking for the end of the array. An alternative method is possible which obviates the necessity for the implicit test. The technique uses an extra entry at the end of the reserved words array. This extra entry guarantees that the search will succeed.

The method uses the extra location to store the string being searched for. The search is then bound to succeed, whether the search string is in the reserved words array proper or not. The extra entry is called a *sentinel* because it stands guard against the possible failure of the test in the new search loop. Figure 14.4 illustrates the sentinel technique.

The sentinel method allows the search loop to be programmed as a repeat-until statement, which never worries about the array index at all:

```
rw[maxrw] := search;
i := 0;
REPEAT
  i := i + 1;
UNTIL search = rw[i];
```

By now you're probably wondering how we tell if the search really succeeds or not. This is so simple, it almost hurts: You failed if you found the sentinel entry; otherwise, you succeeded:

```
lookup := ( i <> maxrw );
```

Maxrw is the index of the sentinel in the reserved words array. If the value of i which causes the loop to exit is

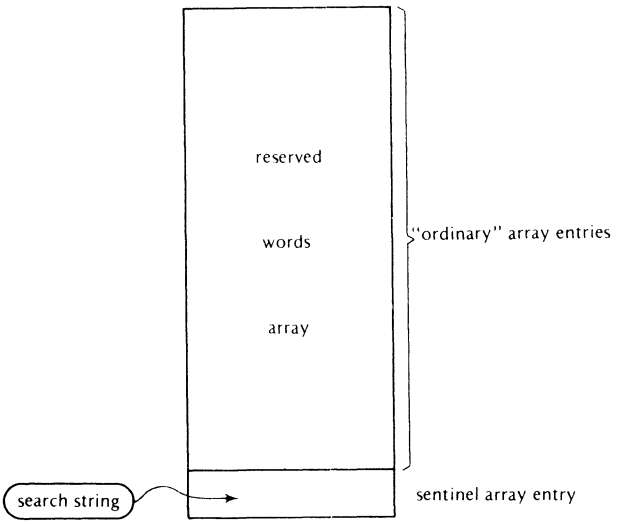


FIGURE 14.4 The Sentinel Technique

anything other than this value, then the search string was found at that index in the rw array. Otherwise, the search only succeeded in finding the sentinel, which is not really succeeding at all.

The value of i only tells the index of the reserved word in the rw array. A separate array is necessary in order to determine the corresponding token value:

```
rwtoken:  ARRAY[1..maxrw] of tokenvalue;
```

Then the assignment:

```
token := rwtoken[i];
```

determines the correct tokenvalue. The rwtoken array must be augmented with a dummy entry corresponding to the sentinel. This entry may contain any value whatsoever, since failure of the lookup function means that the token determined will be idtoken anyway.

For an example of the sentinel technique in action, see Chapter 15.

5. A SKELETON SCANNER

The procedures and functions of Listing 14.2 present a bare-bones implementation of a scanner and its support routines as discussed in this chapter. The main scanner contains only three cases: those for recognizing an integer, for recognizing identifiers or reserved words, and for recognizing the end of input signal. Additional cases must be added in specific applications for handling other token possibilities. These cases should have help from corresponding cases in the classify function, which in the

skeleton implementation recognizes only digits, letters, and the end of input signal character: %.

The getchar routine in this implementation has been written to avoid readsource. This is partially for simplicity and partially since the application of the scanner presented in Chapter 15 is also implemented the same way. The input variable "command" is assumed to be filled by a separate routine. Such a routine would be part of the program which uses the scanner. The scanner may communicate its desire for a new value of "command" by returning the posttoken as the token value.

6. EXPLORATIONS

• Change lookup to return an integer value coded as follows:

−1	Search for reserved word failed.
>0	Search for reserved word succeeded. The value returned represents the index in the rw array of the matching reserved word.

What does this imply about the position in the program where the reserved words array must be declared?

• Write a case for the scanner which distinguishes between the following possible character strings:

<, >, =, <=, >=, <>, =<, =>

What should the corresponding character class that instigates this case be? Make sure you detect erroneous strings as well, such as:

>**<**, ==, **and** <<

• Find examples of search algorithms in your own programs and see whether or not they may be converted to use the sentinel approach. Program the sentinel search in Integer or APPLESOFT BASIC.

• Design a token set for BASIC and see if you can implement a complete scanner for BASIC. What special problems are there which complicate matters?

LISTINGS

LISTING 14.1 EXTRACT WORDS

```
(*PROGRAM testwords;

TYPE

  wordsarray     =        ARRAY[1..100] OF STRING;

VAR

  words:         wordsarray;
  numwords:      INTEGER;

  dissect:       STRING;
  i:             INTEGER;

(*****************************************)
(*                                       *)
(*      e   x   t   r   a   c   t        *)
(*                                       *)
(*   subroutine to extract words from    *)
(* a line of text: a "word" is any       *)
(* sequence of non-blank characters      *)
(* surrounded by blanks or one end of    *)
(* the line.                             *)
(*                                       *)
(*   input:       line - line of text.   *)
(*  output:       numw - number of words *)
(*                      found.           *)
(*                words- words found     *)
(*                       stored in an    *)
(*                       array.          *)
(*                                       *)
(*****************************************)

PROCEDURE extract(

        line:    STRING;
    VAR words:   wordsarray;
    VAR numw:    INTEGER       );

VAR

  sb,cp:         INTEGER;
  nextword:      STRING;

BEGIN

  numw := 0;
  sb   := 0;
  cp   := 1;

  line := concat(line,' ');      (* put IN sentinel *)

  WHILE sb < length(line) DO
```

```
  BEGIN

    WHILE line[cp] <> ' ' DO
      cp := cp + 1;

    nextword := copy(line,sb+1,cp-sb-1);

    IF nextword <> ''
    THEN
    BEGIN

      numw := numw + 1;
      words[numw] := nextword;

    END (* IF nextword <> '' *);

    sb := cp;
    cp := cp + 1;

  END (* WHILE sb < length(line) *);

END (* PROCEDURE extract *);


BEGIN   (***** testwords *****)

  dissect := '';

  REPEAT

    writeln('input a line of text...');
    readln(dissect);

    extract(  dissect, words, numwords  );

    FOR i := 1 TO numwords DO
      writeln(words[i]);

  UNTIL dissect = '';

END.
```

```
(*(*********************************)
(*                                 *)
(*   s k e l e t o n               *)
(*     s c a n n i n g             *)
(*       r o u t i n e s           *)
(*                                 *)
(*  these routines assume the      *)
(* existence of certain types      *)
(* variables -                     *)
(*                                 *)
(*  command,                       *)
(*  tokenbuffer - string or        *)
(*                packed array of  *)
(*                char.            *)
(*  tokenvalue = (idtoken,         *)
(*                numtoken,        *)
(*                posttoken,       *)
(*                etc.             *)
(*  charclass  = (cletter,         *)
(*                cdigit,          *)
(*                cendofinput,     *)
(*                etc.             *)
(*  token - variable of type       *)
(*          tokenvalue - set by    *)
(*          main cases of scan.    *)
(*  etc.                           *)
(*                                 *)
(*********************************)

(*********************************)
(*  g   e   t   c   h   a   r    *)
(*********************************)

FUNCTION getchar: CHAR;
BEGIN

  IF cmdptr > cmdlen
  THEN
    getchar := '%'
    (* END OF input signal *)
  ELSE
  BEGIN
    getchar := command[cmdptr];
    cmdptr  := cmdptr + 1;
  END (*IF*);

END (* FUNCTION getchar *);


(*********************************)
(*  p   u   t   a   n   d   g   e   t   *)
(*********************************)

PROCEDURE putandget;
BEGIN

  IF tokenpos <= 32
  THEN
    tokenbuffer[tokenpos] :=
    nextchar
  (*endif*);

  tokenpos := tokenpos + 1;
  nextchar := getchar;

END (* PROCEDURE putandget *);

(*********************************)
(* s   k   i   p   s   p   a   c   e   s *)
(*********************************)

PROCEDURE skipspaces;
BEGIN

  WHILE nextchar = ' '
  DO
    nextchar := getchar
  (*enddo*);

END (* PROCEDURE skipspaces *);

(*********************************)
(*   l   o   o   k   u   p       *)
(*********************************)

FUNCTION lookup: BOOLEAN;
VAR
  i: INTEGER;

BEGIN

  rw[maxrw] := tokenbuffer;
  i := 0;

  REPEAT
    i := i+ 1;
  UNTIL rw[i] = tokenbuffer;

  token  := rwtoken[i];
  lookup := (i <> maxrw);

END (* FUNCTION lookup *);

(*********************************)
(*   c   o   n   v   e   r   t   *)
(*********************************)
```

```
PROCEDURE convert;
VAR
  value:       INTEGER;

BEGIN

  value := ord(nextchar)-ord('0');

  IF (value<10)
  THEN
    binaryvalue :=
    binaryvalue*10 + value
  ELSE
    writeln(' illegal digit')
  (*endif*);

END (* PROCEDURE convert *);

(*********************************)
(*    c  l  a  s  s  i  f  y     *)
(*********************************)

FUNCTION classify(ch:CHAR):charclass;
BEGIN

  classify := noclass;

  CASE ch OF

    'a','b','c','d','e',
    'f','g','h','i','j',
    'k','l','m','n','o',
    'p','q','r','s','t',
    'u','v','w','x','y',
    'z':

      classify := cletter;

    '0','1','2','3','4',
    '5','6','7','8','9':

      classify := cdigit;

    '%':

      classify := cendofinput;

(*********************************)
(*                               *)
(* insert other cases after this*)
(* comment.  the inserted cases *)
(* will recognize additional     *)
(* classes as needed by the      *)
(* specific PROGRAM IN question.*)
(*                               *)
(*********************************)

  END (* CASE ch OF *);

END (* FUNCTION classify *);

(*********************************)
(*                               *)
(*     s     c     a     n       *)
(*                               *)
(*********************************)

FUNCTION scan: tokenvalue;
BEGIN

  tokenbuffer := blank32;
  tokenpos    := 1;
  token       := notoken;
  skipspaces;

  WHILE token = notoken
  DO
  BEGIN

    CASE classify(nextchar) OF

      cletter:

        BEGIN

          putandget;

          WHILE (nextchar IN letters)
                 OR
                (nextchar IN digits)
          DO
            putandget
          (*enddo*);

          IF NOT lookup
          THEN
            token := idtoken
          (*endif*);

        END;

      cdigit:

        BEGIN

          binaryvalue := 0;
```

LISTING 14.2 (*cont.*)

```
      convert;
      putandget;

      WHILE (nextchar IN digits)
      DO
      BEGIN

        convert;
        putandget;

      END (*DO*);

      token := numtoken;

    END;

  cendofinput:

    BEGIN
      token := posttoken;
    END;

  noclass:

    BEGIN
      writeln('illegal character');
      nextchar := getchar;
      skipspaces;
    END;

  END (*CASE*);

END (*DO*);

scan := token;

END (* FUNCTION scan *);
```

Chapter

15

Interpreters: The Calc Minilanguage

In this chapter, we will present a complete interpreter for a minilanguage called Calc. The purpose of the minilanguage will be to simulate the capabilities of a simple hexadecimal calculator and to provide a couple of simple programmable features at the same time. The program which implements the interpreter will use the techniques of Chapter 14 to decipher the user's input and will also use techniques from the professional world of compiler writers. In particular, you will be introduced to a technique known as *recursive descent*. It is probably the easiest method known for writing a compiler or interpreter short of hiring another programmer. By imitating and adapting this technique, you can implement your own APPLE languages.

1. THE CALC MINILANGUAGE

Calc is a hex calculator simulator implemented as an interpreter. It allows the user to do various arithmetic calculations in one or more bases, including conversion from one base to another. In addition to arithmetic expression evaluation, it allows the user to create identifiers and assign values to them and it allows a single expression to be reevaluated inside a simple loop construct.

The Calc interpreter allows single line commands which are immediately executed. The interpreter prompts the user for a command with a period. Table 15.1 summarizes the commands available in the Calc language.

TABLE 15.1

.HELP	Provides the user with a command summary of the Calc language.
.>expression	Evaluates and prints the value of "expression" using the current output radix. See below for the form of allowable expressions.
.id = expression	Evaluates "expression" and assigns the resulting value to the identifier "id." The id is saved by the interpreter and may be used in subsequent expressions.
.IN base .OUT base	These two commands set the bases which the interpreter will use for input and output of values. Allowable values for base are: HEX, DEC, OCT, BIN, which stand for hexadecimal, decimal, octal, and binary, respectively.
.DUMP	Displays a list of all currently defined identifiers and their values.
.FOR id = num1 TO num2 expression	Repeatedly evaluates "expression" with a different value for id each time. The value of id runs from num1 to num2.

Calc expressions are similar to expressions in most computer languages. The ingredients are as follows:

Identifiers
Numbers (in current input base)
Operators: +, −, *, /, ^
Parentheses: (,)

Some sample expressions are:

```
2 + 2
I + J
3 * (I − OFF * J)
1011 + 1001 + 1111
```

Here is a typical session with Calc. User commands always follow the "." prompt. Calc's responses are on a line not beginning with a ".".

```
WELCOME TO CALC...

HEXADECIMAL
CALCULATOR
SIMULATOR

.IN DEC
.OUT HEX
.10
A
.255
FF
.−1
− MAY NOT APPEAR AT THIS POINT IN THE
COMMAND
.FOR I=10 TO 16 I
A
B
C
D
E
F
10
.IN HEX
.OUT DEC
.0FDED
−531
.800
2048
.FF
MISSING =
MAY NOT APPEAR AT THIS POINT IN THE
COMMAND
.0FF
255
.DUMP
ENTRY:I          VALUE: 17
ENTRY:FF         VALUE:1357
.>I+FF
1374
```

This simple session illustrates a number of points about the implementation of Calc.

- Negative numbers are not allowed as input.
- Output values range between −32768 and 32767.
- Differing input and output bases allows base conversions to be performed.
- Identifiers, once created, are remembered by Calc; in fact, there is no way to get rid of them.
- Hex values need to be preceded by a 0 if they begin with one of the hex digits: A,B,C,D,E, or F.

2. A SCANNER FOR CALC

The Calc language has a very simple token list:

Identifier
Number
+,−,/,*, ^,=,(,)
>

In addition to the identifiers which may be user variables, there are a number of Calc reserved words, each of which yields a separate token:

```
HEX DEC OCT BIN
BYE FOR TO DUMP
IN OUT HELP
```

The scanner for Calc is an elaboration of the scanner presented in Chapter 14. There is a simplification which is worth considering. The routine classify, used in Chapter 14, is not present in the Calc scanner. Instead, the character classification work is coded directly in the scanner main program in the scanner's case statement. This simplification is often desirable when it is not necessary to call the classify routine from within any of the scanner cases proper.

3. THE STRUCTURE OF EXPRESSIONS

The Calc minilanguage and the Calc interpreter can teach us a lot about the nature and processing of expressions in computer languages. In this section we discuss the nesting of expressions and the interpretive evaluation of expressions using a stack.

Nesting of Expressions

Expressions may be nested inside one another. That is, a component of an expression may itself stand alone as a complete expression. For example, the expression:

$$1 / (A + B)$$

contains the subexpressions A, B, 1, and (A + B).

The basic elements making up expressions are operators, operands, and parentheses. The operators tell us what kind of arithmetic operations are to be performed in evaluating an expression. The operands tell us what values to use in evaluating an expression. Finally, the parentheses tell us about the relationship of subexpressions to the whole expression. Parts of an expression enclosed in parentheses are to be evaluated entirely before being used as part of the containing expression. Thus the parentheses in 1/(A + B) tell us to add A and B *before* performing the division.

In order to facilitate the evaluation of expressions by a program, it is common practice to transform expressions either implicitly or explicitly into an alternative notation which avoids the use of parentheses altogether. This notation is known as reverse Polish notation, or RPN, and stems from a notation invented by the Polish mathematician Jan Lukasiewicz and made popular in an academic game called WFF-N-PROOF™. RPN is not as esoteric as it may sound, in fact, its use is required by certain popular handheld electronic calculators such as those sold by Hewlett-Packard.

The trick that allows RPN to avoid the use of parentheses without introducing ambiguity is that subexpressions are written with both operands appearing *before* their corresponding operator. Thus the expression A + B would become AB+ in RPN. The first notation is also called *infix* notation, whereas the latter is known as *postfix* notation. Figure 15.1 shows several expressions written in both parenthesized (infix) and RPN (postfix) form.

The advantage of RPN is that it facilitates the *evaluation* of expressions. This is done using a data structure known as a *stack*. The concept of a stack is much more familiar now than before the advent of microprocessors. Nonetheless we give a brief review before proceeding.

A stack is a data structure similar to an array, but

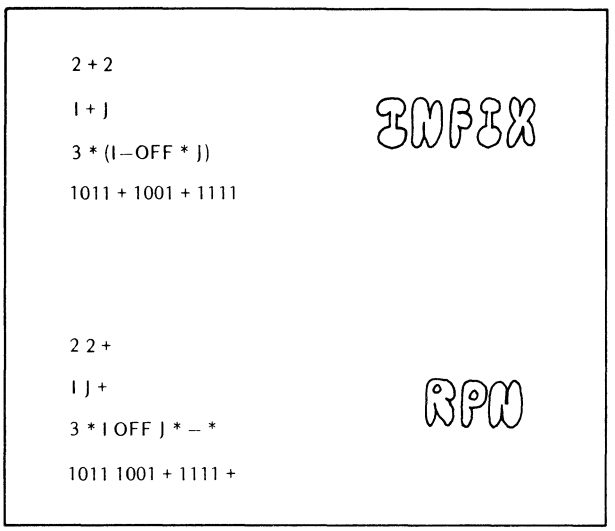


FIGURE 15.1 Expressions in Infix and RPN Form

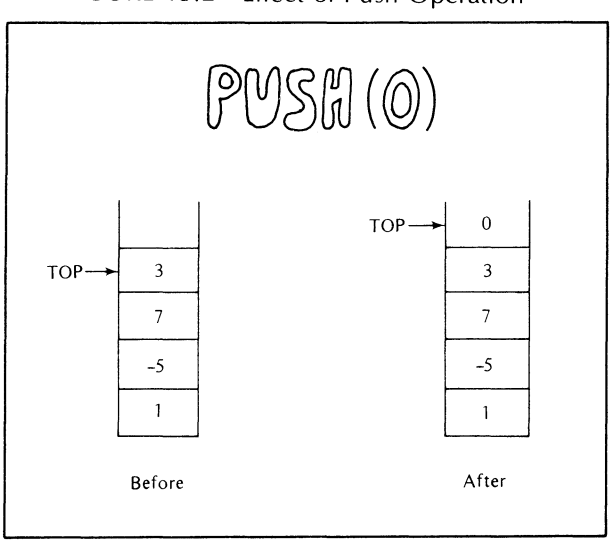
manipulated in a less general fashion. The data contained in a stack is stored and retrieved in LIFO or *Last In, First Out* basis. This is best understood in terms of the simple *operations* possible on a stack. There are PUSH, POP, and TOP.

PUSH	Add an item to the "top" of the stack
POP	Remove an item from the "top" of the stack
TOP	Examine the item at the "top" of the stack without removing it

Figures 15.2 to 15.4 illustrate the effects of these operations.

In order to evaluate expressions using a stack, additional arithmetic operations on the stack must be added to the fundamental PUSH, POP, and TOP operations. In the interpreter presented later, we have added the following

FIGURE 15.2 Effect of Push Operation



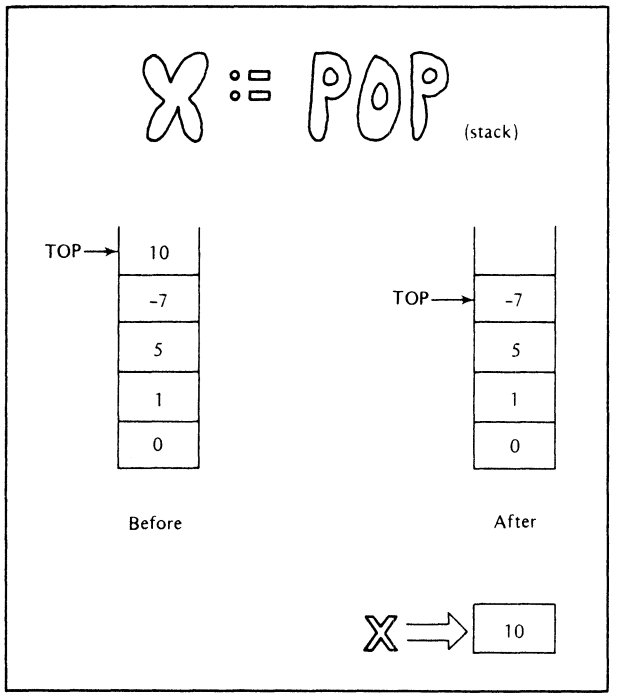
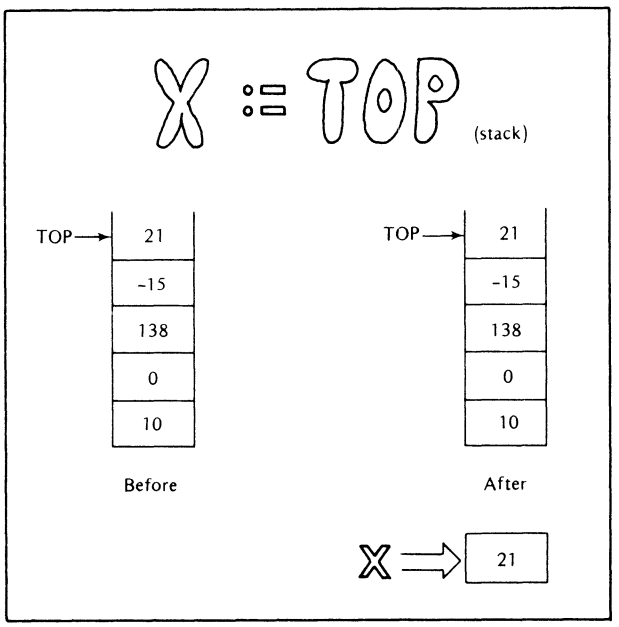


FIGURE 15.3 Effect of Pop Operation

FIGURE 15.4 Effect of Top Operation



stack operations: ADD, SUBTRACT, MULTIPLY, DIVIDE, and EXPONENTIATE.

An arithmetic operation on a stack removes and uses the *top two elements* on the stack as its operands and replaces them with its result. The effect of this is illustrated in Figure 15.5.

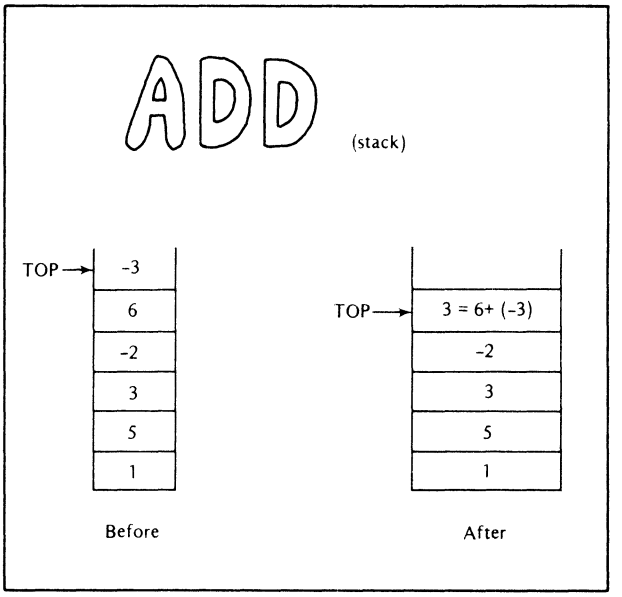


FIGURE 15.5 Add Operation Performed on a Stack

4. PROCESSING EXPRESSIONS AND THE CALC MINILANGUAGE

The main programming example of this chapter is an interpreter for the Calc minilanguage. The structure of this program is detailed in Figure 15.6. The general technique illustrated by the program is a classic one in computer science, known as *recursive descent*.

Recursive Descent

When a user runs the Calc interpreter, a series of one line commands are entered via the keyboard. Each command is translated internally and interpretively executed. Thus, the top level design of the program may be summarized as a loop:

```
while "more commands" do
begin
    getcommand;
    docommand;
end;
```

The Docommand portion of the program parses or validates a command for corrections and then executes it. Most commands are handled by a single separate procedure in the program. For example, there are procedures with names such as help, dump, newradix, and iteration.

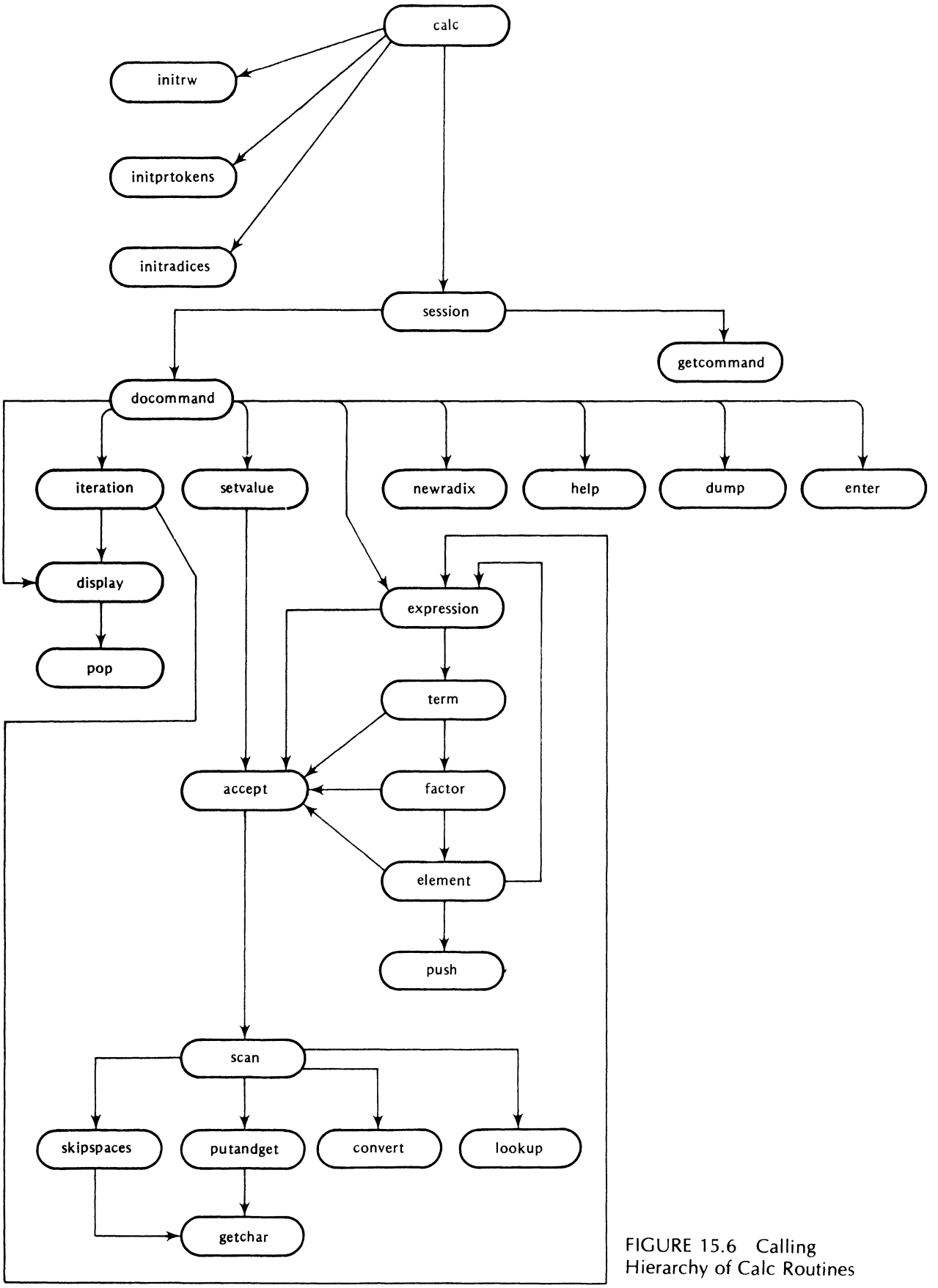


FIGURE 15.6 Calling Hierarchy of Calc Routines

The evaluation of an expression is not quite so simple however. Here the program contains a collection of procedures which must work together in order to interpret an expression. This subcollection is shown separately in Figure 15.7. For each possible component of an expression, there is a separate procedure. These procedures are invoked in a hierarchical or *descending* fashion. So "expression" calls "term," which calls "factor," which calls "element," and so forth. This explains the *descent* half of the term recursive descent.

Earlier, we pointed out that complex expressions may have subexpressions which are full-fledged expressions all by themselves. In order for the procedure expression to evaluate such an expression, it will be necessary for it to implicitly *call itself*. This idea of self-reference is often referred to as *recursion* and provides the explanation of the other half of the terminology. Explicitly, the procedure "element" will call upon the procedure "expression" to recursively evaluate a subexpression. This occurs when element encounters a left parenthesis. Of course, the subexpression might have further subsubexpressions in which case the recursion will extend to more than one level. Recursion can in itself be recursive.

The method of recursive descent models the recursive nature of the expressions in terms of the calling relationship between procedures. This elegant idea is one of the great achievements of early computer programming practices.

5. THE CALC INTERPRETER

The Calc interpreter is shown in Listing 15.1. This program illustrates many techniques used in programming language processors and deserves careful study. We do not have the space to study it exhaustively, but we shall attempt to explain some of the more interesting features.

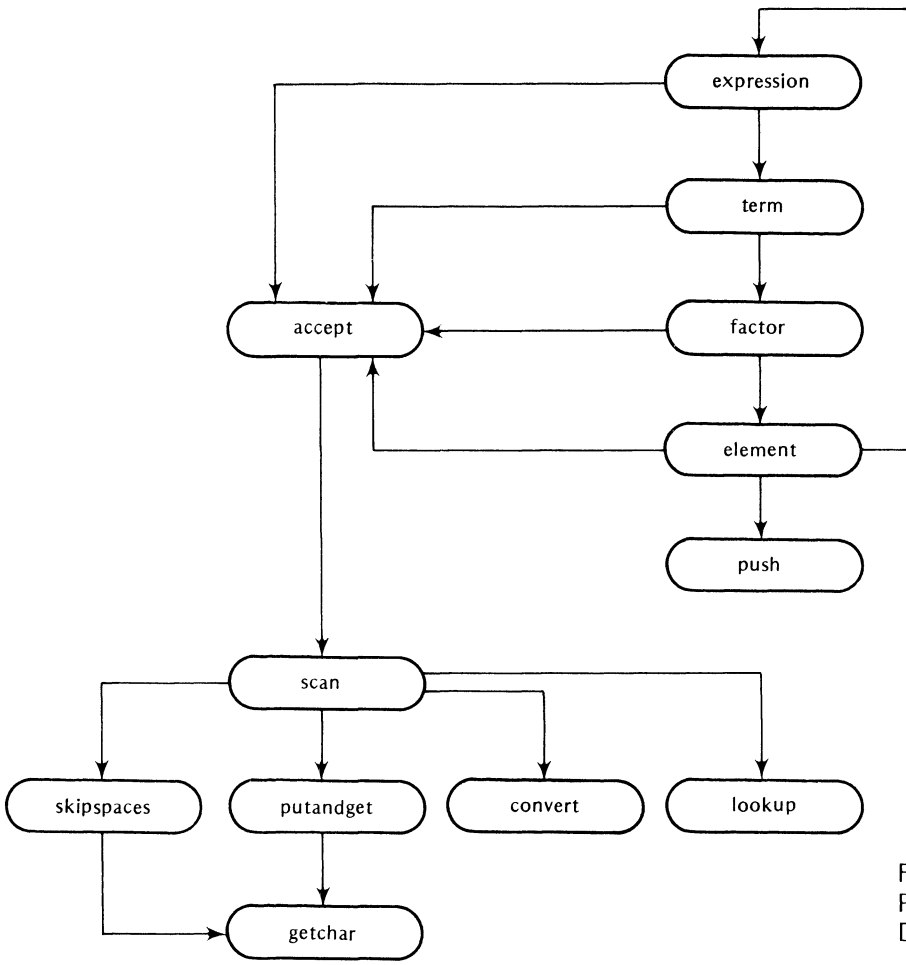


FIGURE 15.7 Expression Processing using Recursive Descent

Syntax Processing—Parsing

In Chapter 14, we discussed the process of scanning or lexical analysis. This process provides a compiler or interpreter with a sequence or stream of tokens, but does nothing to analyze it. An interpreter must verify that a given stream of tokens makes sense structurally, i.e., that it corresponds to a legal language statement. This process is known as *parsing*, and the Calc interpreter cannot shirk its responsibilities in this area.

The recursive descent method is designed for parsing as well as translation. The technique is best explained using examples. To start with, each legal Calc command must *begin* with one of a certain collection of tokens. The legal possibilities are summarized as follows:

token	*statement*
byetoken	.BYE
helptoken	.HELP
dumptoken	.DUMP
intoken	.IN
outtoken	.OUT
evaltoken	.> "expression"
idtoken	.ID = "expression"
fortoken	.FOR ID = ...
numtoken	expression starting with "num"

All other tokens are illegal at the *beginning* of a statement:

```
totoken,plustoken,minustoken, ...
```

The procedure Docommand is responsible for processing this initial token. It accomplishes this using a Pascal case statement:

```
case token of
   byetoken:      ;
   helptoken:     help;
   dumptoken:     dump;
   intoken:       begin
                      accept(intoken);
                      newradix(inr);
                  end;
     ...
   totoken,plustoken,minustoken,...
                  context(token);
end (* case *);
```

There are several points to be made about this statement. First, each legal statement in the Calc language is initiated by a given token. For each given token, there is an individual case to handle it. In each case, this involves calling a support procedure to complete the interpretation of the corresponding statement.

If a statement may contain more than a single token, there must be a mechanism to cause the interpreter to request the scanner to provide it those tokens. In other words, we don't want the interpreter to get stuck looking at the same token over and over again—this would be boring for the interpreter and disastrous for the Calc user. The procedure "accept" is the necessary agent. Its responsibility is to accept the current token, which is passed to it as a parameter, and to call upon the scanner to return the next token.

Now, you might initially protest that accept should not require a parameter. After all, it is simply accepting the *current* token. However, the accept procedure is called in many cases when the interpreter *expects* or *requires* that the current token have a specific value. For example, in the assignment statement:

```
id = expression
```

after the initial id is detected by Calc, it knows that an "=" *must* come next. This means that it may simply pass this information on to accept in order to require such a token. In terms of coding, this saves an IF statement in many situations:

```
accept(equalstoken);
```

as opposed to

```
IF token = equalstoken
THEN
   accept
(*endif*);
```

This is a matter of coding style more than a difference in the algorithm being used. One final point about accept: if accept finds that its argument does *not* match the current token, it will merely announce that the token is missing and not call scan for a new token. The statement

```
.I 10
```

will be interpreted by Calc *as if it had been* .I = 10. That is, the accept procedure will simulate the *insertion* of the missing "=" in the original statement.

When Calc is processing a portion of a statement at which a choice of tokens is possible, it will use a case statement, as in the Docommand procedure. The tokens which are *not* legal in that situation will cause the procedure "context" to be called. Context effectively *discards*

the offending token, while notifying the user by printing the message:

MAY NOT APPEAR AT THIS POINT IN THE COMMAND.

This allows the interpreter to recover from poisonous statements such as:

```
.I = = 10
```

or

```
.5 + + 5
```

6. EXPLORATIONS

- Investigate whether or not the procedure "element" may be recoded so that the statement .I = = 10 would be processed as if it were .I = 10.
- Can you extend the Calc input technique so that it accepts multiple line commands?
- Add to the HELP facility of CALC. Extend it to provide more information on individual commands; see Chapter 8.
- Implement your own minilanguage using the techniques of this chapter.

LISTINGS

LISTING 15.1 HEX CALCULATOR SIMULATOR

```
(*(*$s+*)
PROGRAM calc;

(*************************************************************************)
(*                                                                       *)
(*                              c    a    l    c                        *)
(*                                                                       *)
(*  . this program provides the facilities of a simulated hand-held     *)
(* calculator.  it accepts one line commands which may specify the       *)
(* evaluation of expressions involving variables, conversion of values   *)
(* from one base to another, etc.                                        *)
(*    the commands fed to the program are translated by the method of    *)
(* recursive descent.  in this method, each grammatical construct that   *)
(* may be used in a legal statement is translated by a single procedure. *)
(* since many constructs involve other constructs, e.g. expressions      *)
(* involve identifiers, the recognition procedures must invovke one      *)
(* another.  in general, a given procedure may be invoked recursively    *)
(* through a series of calls: hence, the name for the method.            *)
(*                                                                       *)
(*************************************************************************)


CONST

  maxrw          =       12;
  maxrwlen       =       3;
  blank32        =       '                                ';
  home           =       12;
  clreol         =       29;


TYPE

  tokenvalue     =

    (notoken,idtoken,numtoken,
     plustoken,minustoken,
     multtoken,divtoken,exptoken,
     evaltoken,equalstoken,
     lparen,rparen,hextoken,
     dectoken,octtoken,bintoken,
     byetoken,posttoken,fortoken,
     totoken,dumptoken,intoken,
     outtoken,helptoken);

  bases          =

    (binary,octal,
     decimal,hexadecimal);

  radixkind    =    (inr,outr);
```

LISTING 15.1 *(cont.)*

```
  entry       =   ^idrec;

  idrec = RECORD

            id:     STRING[32];
            value:  INTEGER;
            next:   entry;

          END;

  stack = RECORD

            top:    INTEGER;
            value:  ARRAY[1..25]
                    OF INTEGER;

          END;


VAR

  rw:         ARRAY[1..maxrw]
              OF STRING[32];
  rwtoken:    ARRAY[1..maxrw]
              OF tokenvalue;
  prtoken:
    ARRAY[notoken..helptoken]
    OF STRING[32];

  command:    STRING[80];
  cmdlen:     1..80;
  cmdptr:         INTEGER;
  tokenbuffer:    STRING[32];
  savetoken:      STRING[32];
  token:          tokenvalue;
  oldtoken:       tokenvalue;
  nextchar:       CHAR;
  tokenpos:       INTEGER;
  exprstart:      INTEGER;
  loopstart:      INTEGER;
  loopfinish:     INTEGER;
  letters:        SET OF CHAR;
  digits:         SET OF CHAR;
  hexdigits:      SET OF CHAR;
  inradix:        INTEGER;
  outradix:       INTEGER;
  radices:
    ARRAY[binary..hexadecimal]
    OF INTEGER;
  base:           bases;
  binaryvalue:    INTEGER;
  prdigit:        INTEGER;
  first:          entry;
  sentinel:       entry;
  newentry:       entry;
  entryfound:     entry;
  lhs:            entry;
  loopcontrol:    entry;
  valstack:       stack;
  digitstack:     stack;

PROCEDURE push(VAR s:stack;
                   v:INTEGER); FORWARD;
PROCEDURE docommand;           FORWARD;
PROCEDURE help;                FORWARD;
PROCEDURE expression;          FORWARD;
PROCEDURE iteration;           FORWARD;
PROCEDURE setvalue;            FORWARD;
PROCEDURE accept(
              t:tokenvalue);   FORWARD;
PROCEDURE term;                FORWARD;
PROCEDURE factor;              FORWARD;
PROCEDURE element;             FORWARD;
PROCEDURE display;             FORWARD;

(*******************************)
(*   i   n   i   t   r   w     *)
(*******************************)

PROCEDURE initrw;
BEGIN

  rw[1] := 'hex                    ';
  rwtoken[1] := hextoken;
  rw[2] := 'dec                    ';
  rwtoken[2] := dectoken;
  rw[3] := 'oct                    ';
  rwtoken[3] := octtoken;
  rw[4] := 'bin                    ';
  rwtoken[4] := bintoken;
  rw[5] := 'bye                    ';
  rwtoken[5] := byetoken;
  rw[6] := 'FOR                    ';
  rwtoken[6] := fortoken;
  rw[7] := 'TO                     ';
  rwtoken[7] := totoken;
  rw[8] := 'dump                   ';
  rwtoken[8] := dumptoken;
  rw[9] := 'IN                     ';
  rwtoken[9] := intoken;
  rw[10] := 'out                    ';
  rwtoken[10] := outtoken;
  rw[11] := 'help                   ';
  rwtoken[11] := helptoken;
  rw[maxrw] := blank32;
  rwtoken[maxrw] := notoken;
```

```
END (* PROCEDURE initrw *);

(*********************************)
(*   i n i t p r t o k e n s   *)
(*********************************)

PROCEDURE initprtokens;
BEGIN

  prtoken[notoken]     :=
              'notoken';
  prtoken[idtoken]     :=
              'identifier';
  prtoken[numtoken]    :=
              'number';
  prtoken[plustoken]   := '+';
  prtoken[minustoken]  := '-';
  prtoken[multtoken]   := '*';
  prtoken[divtoken]    := '/';
  prtoken[exptoken]    := '^';
  prtoken[evaltoken]   := '>';
  prtoken[equalstoken]:= '=';
  prtoken[lparen]      := '(';
  prtoken[rparen]      := ')';
  prtoken[hextoken]    := 'hex';
  prtoken[dectoken]    := 'dec';
  prtoken[octtoken]    := 'oct';
  prtoken[bintoken]    := 'bin';
  prtoken[byetoken]    := 'bye';
  prtoken[posttoken]   := '%';
  prtoken[fortoken]    := 'FOR';
  prtoken[totoken]     := 'TO';
  prtoken[intoken]     := 'IN';
  prtoken[outtoken]    := 'out';
  prtoken[helptoken]   := 'help';

END (* PROCEDURE initprtokens *);

(*********************************)
(*   i n i t r a d i c e s     *)
(*********************************)

PROCEDURE initradices;
BEGIN

  radices[binary]      := 2;
  radices[octal]       := 8;
  radices[decimal]     := 10;
  radices[hexadecimal]:= 16;

END (*PROCEDURE initradices *);

(*********************************)
(*     s   c   r   e   e   n     *)
(*********************************)

PROCEDURE screen(f: INTEGER);
BEGIN

  writeln( chr(f));

END (* PROCEDURE screen *);

(*********************************)
(*     e     n     t     e     r     *)
(*********************************)

FUNCTION enter: BOOLEAN;
VAR
  search:      entry;

BEGIN
  sentinel^.id := tokenbuffer;
  search        := first;

  WHILE search^.id <> tokenbuffer
  DO
    search := search^.next
  (*enddo*);

  IF search = sentinel
  THEN
  BEGIN
    new(newentry);
    newentry^.id := tokenbuffer;
    newentry^.next := first;
    first           := newentry;
    entryfound    := newentry;
  END
  ELSE
    entryfound    := search
  (*endif*);

  enter := (search <> sentinel);

END (* FUNCTION enter *);

(*********************************)
(*        d    u    m    p          *)
(*********************************)

PROCEDURE dump;
VAR
  i:  INTEGER;
  de: entry;
```

LISTING 15.1 (*cont.*)

```
BEGIN
  de := first;

  WHILE de <> sentinel
  DO
  BEGIN
    write('entry: ');
    i := 1;
    WHILE de^.id[i] <> ' '
    DO
    BEGIN
      write(de^.id[i]);
      i := i + 1;
    END (*DO*);
    write('     value: ');
    push(valstack,de^.value);
    display;
    de := de^.next;
  END;

END (* PROCEDURE dump *);

(*********************************)
(*  c   o   n   t   e   x   t  *)
(*********************************)

PROCEDURE context( t: tokenvalue );
BEGIN

  IF t=notoken
  THEN
    writeln('illegal command')
  ELSE
  BEGIN
    write(prtoken[token]);
    writeln(' may not appear at
    this point in the command');
    accept(token);
  END (*IF*);

END (* PROCEDURE context *);

(*********************************)
(*   n  e  w  r  a  d  i  x     *)
(*********************************)

PROCEDURE newradix(r:radixkind);

  PROCEDURE newbase;
  BEGIN

    CASE token OF

      hextoken:
          base := hexadecimal;
      dectoken:
          base := decimal;
      octtoken:
          base := octal;
      bintoken:
          base := binary;

      notoken,idtoken,numtoken,
      plustoken,minustoken,
      multtoken,divtoken,
      exptoken,evaltoken,
      equalstoken,lparen,rparen,
      byetoken,posttoken,
      fortoken,totoken,
      dumptoken,intoken,outtoken:

        context(token);

    END (*CASE*);

  END (* PROCEDURE newbase *);

BEGIN (* newradix *)

  newbase;

  IF r=inr
  THEN
    inradix  := radices[base]
  ELSE
    outradix := radices[base]
  (*endif*);

END (* PROCEDURE newradix *);

(*********************************)
(*      p   u   s   h            *)
(*********************************)

PROCEDURE push;
BEGIN

  WITH s
  DO
  BEGIN
    top := top + 1;
    value[top] := v;
  END (*WITH*);

END (* PROCEDURE push *);
```

```
(*******************************)
(*       p   o   p             *)
(*******************************)

FUNCTION pop(VAR s:stack):INTEGER;
BEGIN

  WITH s
  DO
  BEGIN
    pop := value[top];
    top := top - 1;
  END (*WITH*);

END (* FUNCTION pop *);

(*******************************)
(*  d   i   s   p   l   a   y  *)
(*******************************)

PROCEDURE display;
BEGIN

  binaryvalue := pop(valstack);
  digitstack.top := 0;

  IF binaryvalue < 0
  THEN
  BEGIN
    binaryvalue := -binaryvalue;
    write('-');
  END (* IF *);

  REPEAT

    prdigit :=
      binaryvalue MOD outradix;
    binaryvalue :=
      binaryvalue DIV outradix;
    push(digitstack,prdigit);

  UNTIL binaryvalue = 0;

  REPEAT

    prdigit := pop(digitstack);

    CASE prdigit OF

      0: write('0');
      1: write('1');
      2: write('2');
      3: write('3');
      4: write('4');
      5: write('5');
      6: write('6');
      7: write('7');
      8: write('8');
      9: write('9');
      10:write('a');
      11:write('b');
      12:write('c');
      13:write('d');
      14:write('e');
      15:write('f');

    END (* CASE *);

  UNTIL digitstack.top = 0;
  writeln;

end (* procedure display *);

(*******************************)
(*  o   p   e   r   a   t   e  *)
(*******************************)

PROCEDURE operate(t:tokenvalue);
VAR
  tos:          INTEGER;
  nos:          INTEGER;
  result:       INTEGER;
  i:            INTEGER;

BEGIN

  tos := pop(valstack);
  nos := pop(valstack);

  CASE t OF

    plustoken:
      result := tos+nos;
    minustoken:
      result := nos-tos;
    multtoken:
      result := tos*nos;
    divtoken:
      result := nos DIV tos;
    exptoken:
      BEGIN
        result := nos;
        i := 2;
        WHILE i <= tos
        DO
```

```
            BEGIN
              result := result*nos;
              i := i + 1;
            END (*DO*);
          END;

      END (*CASE*);
      push(valstack,result);

    END (* PROCEDURE operate *);

    (*********************************)
    (*     g e t c o m m a n d       *)
    (*********************************)

    PROCEDURE getcommand;
    BEGIN

      REPEAT

        write('.');
        readln(command);

      UNTIL length(command)>0;

      cmdlen := length(command);
      cmdptr := 1;
      nextchar := ' ';      (*force first getchar*)

    END (* PROCEDURE getcommand *);

    (*$icalc.scan.text*)

    (*$icalc.main.text*)

  (*(*(*******************************)
  (*  g   e   t   c   h   a   r  *)
  (*******************************)

  FUNCTION getchar: CHAR;
  BEGIN

    IF cmdptr > cmdlen
    THEN
      getchar := '%'
    ELSE
    BEGIN
      getchar := command[cmdptr];
      cmdptr  := cmdptr + 1;
    END (*IF*);

  END (* FUNCTION getchar *);
```

```
(*********************************)
(*  p u t a n d g e t  *)
(*********************************)

PROCEDURE putandget;
BEGIN

  IF tokenpos <= 32
  THEN
    tokenbuffer[tokenpos] :=
      nextchar
  (*endif*);

  tokenpos := tokenpos + 1;
  nextchar := getchar;

END (* PROCEDURE putandget *);

(*********************************)
(*    s k i p s p a c e s      *)
(*********************************)

PROCEDURE skipspaces;
BEGIN

  WHILE nextchar = ' '
  DO
    nextchar := getchar
  (*enddo*);

END (* PROCEDURE skipspaces *);

(*********************************)
(*   l   o   o   k   u   p     *)
(*********************************)

FUNCTION lookup: BOOLEAN;
VAR
  i: INTEGER;

BEGIN

  rw[maxrw] := tokenbuffer;
  i := 0;

  REPEAT
    i := i+ 1;
  UNTIL rw[i] = tokenbuffer;

  token  := rwtoken[i];
  lookup := (i <> maxrw);

END (* FUNCTION lookup *);
```

```
(*********************************)
(*  c   o   n   v   e   r   t  *)
(*********************************)

PROCEDURE convert;
VAR
  value:       INTEGER;

BEGIN
  IF nextchar IN hexdigits
  THEN
    value :=
      ord(nextchar)-ord('a')+10
  ELSE
    value :=
      ord(nextchar)-ord('0')
  (*endif*);

  IF (value<inradix)
  THEN
    binaryvalue :=
      binaryvalue*inradix + value
  ELSE
    writeln('digit illegal
             for current radix')
  (*endif*);

END (* PROCEDURE convert *);

(*********************************)
(*                               *)
(*   s     c     a     n       *)
(*                               *)
(*********************************)

FUNCTION scan: tokenvalue;
BEGIN

  tokenbuffer := blank32;
  tokenpos    := 1;
  token       := notoken;
  skipspaces;

  WHILE token = notoken
  DO
  BEGIN

    CASE nextchar OF

    'a','b','c','d','e',
    'f','g','h','i','j',
    'k','l','m','n','o',
```

LISTING 15.1 (*cont.*)

```
'p','q','r','s','t',
'u','v','w','x','y',
'z':

BEGIN

  putandget;

  WHILE (nextchar IN letters)
         OR
        (nextchar IN digits)
  DO
    putandget
  (*enddo*);

  IF NOT lookup
  THEN
    token := idtoken
  (*endif*);

END;

'0','1','2','3','4',
'5','6','7','8','9':

BEGIN

  binaryvalue := 0;
  convert;
  putandget;

  WHILE (nextchar IN digits) OR
        (nextchar IN hexdigits)
  DO
  BEGIN

    convert;
    putandget;

  END (*DO*);

  token := numtoken;

END;

'+':

BEGIN
  token := plustoken;
  putandget;
END;

'-':

BEGIN
  token := minustoken;
  putandget;
END;

'*':

BEGIN
  token := multtoken;
  putandget;
END;

'/':

BEGIN
  token := divtoken;
  putandget;
END;

'^':

BEGIN
  token := exptoken;
  putandget;
END;

'>':

BEGIN
  token := evaltoken;
  putandget;
END;

'=':

BEGIN
  token := equalstoken;
  putandget;
END;

'(':

BEGIN
  token := lparen;
  putandget;
END;

')':

BEGIN
  token := rparen;
  putandget;
```

```
    END;

    '%':

    BEGIN
      token := posttoken;
    END;

    END (*CASE*);

    IF token = notoken
    THEN
    BEGIN
      writeln('illegal character');
      nextchar := getchar;
      skipspaces;
    END (*IF*);

  END (*DO*);

  scan := token;

END (* FUNCTION scan *);

(*******************************)
(*      a  c  c  e  p  t       *)
(*******************************)

PROCEDURE accept;
BEGIN

  IF token = t
  THEN
    token := scan
  ELSE
  BEGIN
    write('missing ');
    writeln(prtoken[t]);
  END (*IF*);

END (* FUNCTION accept *);

(*(*(*****************************)
(*  s   e   s   s   i   o   n  *)
(*****************************)

PROCEDURE session;
BEGIN
  REPEAT

    token := notoken;
    getcommand;
    docommand;

  UNTIL token = byetoken;

END (* PROCEDURE session *);

(*******************************)
(*  d  o  c  o  m  m  a  n  d  *)
(*******************************)

PROCEDURE docommand;
BEGIN

  valstack.top := 0;
  IF token = notoken
  THEN
    accept(notoken)
  (*endif*);

  CASE token OF

    byetoken:          ;

    helptoken:    help;

    dumptoken:    dump;

    intoken:
      BEGIN
        accept(intoken);
        newradix(inr);
      END;

    outtoken:
      BEGIN
        accept(outtoken);
        newradix(outr);
      END;

    evaltoken:
      BEGIN
        accept(evaltoken);
        expression;
        display;
      END;

    idtoken:
      BEGIN
        IF
          enter
        THEN
        (*endif*);
        lhs := entryfound;
        accept(idtoken);
```

LISTING 15.1 *(cont.)*

```
        setvalue;
      END;

    fortoken:
      BEGIN
        accept(fortoken);
        iteration;
      END;

    numtoken:
      BEGIN
        expression;
        display;
      END;

    totoken,plustoken,minustoken,
    multtoken,divtoken,
    exptoken,equalstoken,lparen,
    rparen,hextoken,dectoken,
    octtoken,bintoken:

      context(token);

  END (*CASE*);

END (* PROCEDURE docommand *);

(*********************************)
(*      h   e   l   p            *)
(*********************************)

PROCEDURE help;
BEGIN

  writeln('the legal commands are
           as follows:');
  writeln;
  writeln('in (hex,dec,bin)');
  writeln('out (hex,dec,bin)');
  writeln('>expression');
  writeln('expression');
  writeln('id = expression');
  writeln('dump');
  writeln('for id = num to num
           expression');
  writeln('help');

END (* PROCEDURE help *);

(*********************************)
(*   s   e   t   v   a   l   u   e   *)
(*********************************)

PROCEDURE setvalue;
BEGIN

  accept(equalstoken);
  expression;
  lhs^.value := pop(valstack);

END (* PROCEDURE setvalue *);

(*********************************)
(*    e x p r e s s i o n        *)
(*********************************)

PROCEDURE expression;
VAR
  done:        BOOLEAN;

BEGIN
  term;
  done := false;

  REPEAT

    IF token = plustoken
    THEN
    BEGIN
      accept(plustoken);
      term;
      operate(plustoken);
    END
    ELSE
      IF token = minustoken
      THEN
      BEGIN
        accept(minustoken);
        term;
        operate(minustoken);
      END
      ELSE
        done := true
      (*endif*)
    (*endif*);

  UNTIL done;

END (* PROCEDURE expression *);

(*********************************)
(*        t   e   r   m          *)
(*********************************)

PROCEDURE term;
VAR
```

```
  done:        BOOLEAN;

BEGIN

  factor;
  done := false;

  REPEAT

    IF token = multtoken
    THEN
    BEGIN
      accept(multtoken);
      factor;
      operate(multtoken);
    END
    ELSE
      IF token = divtoken
      THEN
      BEGIN
        accept(divtoken);
        factor;
        operate(divtoken);
      END
      ELSE
        done := true
      (*endif*)
    (*endif*);

  UNTIL done;

END (* PROCEDURE term *);

(*********************************)
(*    f    a    c    t    o    r      *)
(*********************************)

PROCEDURE factor;
VAR
  done:        BOOLEAN;

BEGIN

  element;
  done := false;

  REPEAT

    IF token = exptoken
    THEN
    BEGIN
      accept(exptoken);
      element;
      operate(exptoken);
    END
    ELSE
      done := true
    (*endif*);

  UNTIL done;

END (* PROCEDURE factor *);

(*********************************)
(*  e   l   e   m   e   n   t  *)
(*********************************)

PROCEDURE element;
BEGIN


  IF token = idtoken
  THEN
  BEGIN
    IF NOT enter
    THEN
      writeln('undefined id
                in expression')
    (*endif*);
    push(valstack,
         entryfound^.value);
    token := scan;
  END
  ELSE
    IF token = numtoken
    THEN
    BEGIN
      push(valstack,
           binaryvalue);
      accept(numtoken);
    END
    ELSE
      IF token = lparen
      THEN
      BEGIN
        accept(lparen);
        expression;
        accept(rparen);
      END (*THEN*)
      ELSE
```

LISTING 15.1 *(cont.)*

```
      BEGIN
        push(valstack,0);
        context(token);
      END (*IF*);
    (*endif*)
  (*endif*);

END (* PROCEDURE element *);

(*********************************)
(*  i  t  e  r  a  t  i  o  n  *)
(*********************************)

PROCEDURE iteration;
VAR
  ok: BOOLEAN;

BEGIN
  ok := true;
  IF token = idtoken
  THEN
  BEGIN
    IF enter THEN;
    loopcontrol := entryfound;
    accept(idtoken);
  END
  ELSE
  BEGIN
    writeln('missing control
            variable in do command');
    ok := false;
  END;
  IF ok
  THEN
  BEGIN
    accept(equalstoken);
    accept(numtoken);
    loopstart := binaryvalue;
    accept(totoken);
    exprstart := cmdptr;
    accept(numtoken);
    loopfinish := binaryvalue;
    loopcontrol^.value := loopstart;
    expression;
    display;
    loopcontrol^.value :=
      loopcontrol^.value + 1;
    WHILE loopcontrol^.value <=
          loopfinish
    DO
    BEGIN
      cmdptr := exprstart;
      nextchar := ' ';
      (* force getchar *)
      IF token = posttoken
      THEN
        accept(posttoken)
      (*endif*);
      expression;
      display;
      loopcontrol^.value :=
        loopcontrol^.value + 1;
    END;
  END (*IF ok*);

END (* PROCEDURE iteration *);

(*********************************)
(*                               *)
(*  m    m aaaaa iiiii n    n   *)
(*  mm mm a    a    i    nn   n   *)
(*  m m m a    a    i    n n  n   *)
(*  m m m aaaaa     i    n  n n   *)
(*  m    m a    a    i    n   nn  *)
(*  m    m a    a    i    n    n  *)
(*  m    m a    a iiiii n    n   *)
(*                               *)
(*********************************)

BEGIN

  letters   := ['a'..'z'];
  digits    := ['0'..'9'];
  hexdigits := ['a'..'f'];

  inradix   := 10;
  outradix  := 10;
  base      := decimal;

  initrw;
  initprtokens;
  initradices;

  new(sentinel);
  sentinel^.next := NIL;
  sentinel^.value := 0;
  first          := sentinel;
```

```
  screen(home);
  writeln('welcome to calc...');
  writeln;
  writeln('  h e x a d e c i m a l ');
  writeln('  c a l c u l a t o r   ');
  writeln('  s i m u l a t o r     ');
  writeln;
  writeln;

  session;

end.
```

Chapter

16

Using Pascal Units

If you plan on writing many programs in APPLE Pascal, then you should seriously consider learning how to use Units. Units allow you to collect reusable routines into little personal libraries. The routines may be compiled as a separate package. The code file produced may then be used as an auxiliary library or the routines installed in the SYSTEM.LIBRARY. In this chapter, we present some example Units and demonstrate their use.

1. A VERY BRIEF REVIEW OF UNITS

The APPLE Pascal Reference Manual gives a good introduction to the structure and use of Units. The following diagram should serve to summarize the syntax and layout of a plain Unit. (We shall not deal with separate units or intrinsic units here.)

UNIT blah-de-blah;

INTERFACE (* The "public" part of a UNIT *)

const, type, var, procedure, and function *declarations*.

IMPLEMENTATION (* The "private" part of a UNIT *)

const, type, var declarations. procedure and function implementations to match those declared in the INTERFACE. procedure and function implementations that are totally local and private to the unit.

BEGIN

Initialization section. This code gets executed when the Unit is loaded. This enables variables to be given values, etc.

END.

2. THE DIET GRAPH PROGRAM

The program of Listing 16.1 provides a simple personal application. It constructs a graph showing the expected weight loss curve (by day) for a person on a caloric deprivation diet. Listing 16.2 shows the output from a sample run of the program.

There are four routines used by the Diet Graph program which could easily be reused in other programs. They are:

PROCEDURE writedate(thisdate: date); Outputs a day in month,day format.

PROCEDURE tomorrow(var today:date); Calculate the day following today.

PROCEDURE initmonths; Set up arrays containing information about months.

PROCEDURE datein(var thisdate: date); Accept a date from CONSOLE:.

Supporting these routines are two types and several variables:

```
TYPE
  months = (january,february,march,april,may,
            june,july,august,september,
            october,november,december);
  date   = RECORD
             mon:    months;
             day:    1..31;
           END;
VAR
  elapsed:      ARRAY[january..december]
                OF INTEGER;
  daysinmonth:  ARRAY[january..december]
                OF INTEGER;
  monthnames:   ARRAY[january..december]
                OF STRING[3];
  invert:       ARRAY[1..12] OF months;
```

Listing 16.3 presents a Unit composed of the four routines and their supporting types and variables. Then Listing 16.4 shows a revised version of the Diet Graph program which uses the Calendar Unit of Listing 16.3.

The program of Listing 16.4 requests the Calendar routines with a USES statement:

```
USES calendar;
```

Normally, a USES will cause the compiler to search for a Unit in the SYSTEM.LIBRARY. In Listing 16.4, we have made use of the U option to the APPLE Pascal compiler. The line:

```
(*$Ucalendar.code*)
```

informs the compiler that the file CALENDAR.CODE is to be treated as a user program library. Then the USES calendar statement causes the compiler to search the file CALENDAR.CODE for the Calendar Unit.

3. A LOW-RESOLUTION GRAPHICS UNIT

APPLE Pascal does not provide a means of accessing the APPLE II's low-resolution graphics. Listing 16.5 is a step toward remedying this situation. It provides procedures written in Pascal equivalent to the BASIC statements:

```
HLIN, VLIN, PLOT
COLOR, GR, TEXT, SCREEN
```

Note that the procedure GR is actually equivalent to the sequence of BASIC statements:

```
GR
POKE -16302,0
COLOR=0
FOR I=40 TO 47: HLIN 0,39 AT I: NEXT I
```

The Lowres Unit makes use of another unit called "peekpoke." This is presented in Listing 16.6 and provides a Pascal procedure and function which are equivalent to the BASIC statements POKE and PEEK.

The low-resolution routines implemented in this fashion are quite slow when compared to the corresponding BASIC statements. This somewhat limits their usefulness. Still they may be used to add simple drawing capabilities to Pascal programs. Listing 16.7 presents an application of the Lowres Unit which draws a picture of a house.

4. EXPLORATIONS

• Investigate your own Pascal programs. Are there collections of routines which you could put into a Unit? If so, do it!

LISTINGS

LISTING 16.1 DIET GRAPH PROGRAM

```
(*(*************************************)
(*                                     *)
(*     D  i  e  t  g  r  a  p  h       *)
(*                                     *)
(*     This program constructs a bar   *)
(* graph which represents the expect-  *)
(* weight loss during a caloric de-    *)
(* privation diet.  The loss is cal-   *)
(* culated based on the following      *)
(* assumptions:                        *)
(*                                     *)
(*   1. A pound of fat (body weight    *)
(*      lost first) is equivalent to   *)
(*      3500 calories.                 *)
(*   2. The body requires 15 calories  *)
(*      per pound to sustain a given   *)
(*      weight level.                  *)
(*   3. The dieter will adhere to a    *)
(*      regimen of so many calories    *)
(*      per day for the duration of    *)
(*      the projected dieting period.  *)
(*                                     *)
(*************************************)

PROGRAM dietgraph;
CONST

  poundoffat     =        3500.0;
  calsperlb      =          15.0;
  enhance        =             1;
  normal         =             2;

  tiny           =            31;
  small          =            30;
  medium         =            29;
  large          =            28;

TYPE

  months   =   (january,february,march,april,
                may,june,july,august,september,
                october,november,december);

  date   =   RECORD

               mon:        months;
               day:        1..31;

             END;

VAR
```

```
  elapsed:       ARRAY[january..december] OF INTEGER;
  daysinmonth:   ARRAY[january..december] OF INTEGER;
  monthnames:    ARRAY[january..december] OF STRING[3];
  invert:        ARRAY[1..12] OF months;

  lst:                    text;

  name:                   STRING;

  barweight:     ARRAY[1..365] OF REAL;

  start:                  REAL;
  calperday:              REAL;
  deprivation:            REAL;
  dailyweight:            REAL;
  loss:                   REAL;

  stars:                  INTEGER;
  i:                      INTEGER;
  dateday:                date;
  thisday:                INTEGER;
  lastday:                INTEGER;
  lowscale:               INTEGER;
  next:                   INTEGER;
  maintain:               INTEGER;

(*****************************************)
(*     w  r  i  t  e  d  a  t  e         *)
(*****************************************)

PROCEDURE writedate(thisdate: date);
BEGIN

  write(lst,monthnames[thisdate.mon] );
  write(lst,thisdate.day:3);

END (* PROCEDURE writedate *);

(*****************************************)
(*   t   o   m   o   r   r   o   w       *)
(*****************************************)

PROCEDURE tomorrow(VAR today:date);
VAR
  i:  INTEGER;
  d:  0..31;
  m:  months;

BEGIN

  m := today.mon;
  d := today.day;
```

LISTING 16.1 *(cont.)*

```
  IF d+1 > daysinmonth[m]
  THEN
  BEGIN

    today.day := 1;

    IF today.mon = december
    THEN
      today.mon := january
    ELSE
      today.mon := succ(today.mon)
    (* endif *);

  END (* IF d+1> ... *)
  ELSE
    today.day := today.day + 1
  (* endif *);

END (* PROCEDURE tomorrow *);

(*****************************************)
(*    i  n  i  t  m  o  n  t  h  s      *)
(*****************************************)

PROCEDURE initmonths;
VAR
  i:  INTEGER;
  m:  months;
BEGIN

  monthnames[january]   := 'jan';
  monthnames[february]  := 'feb';
  monthnames[march]     := 'mar';
  monthnames[april]     := 'apr';
  monthnames[may]       := 'may';
  monthnames[june]      := 'jun';
  monthnames[july]      := 'jul';
  monthnames[august]    := 'aug';
  monthnames[september] := 'sep';
  monthnames[october]   := 'oct';
  monthnames[november]  := 'nov';
  monthnames[december]  := 'dec';

  daysinmonth[january]   := 31;
  daysinmonth[february]  := 28;
  daysinmonth[march]     := 31;
  daysinmonth[april]     := 30;
  daysinmonth[may]       := 31;
  daysinmonth[june]      := 30;
  daysinmonth[july]      := 31;
  daysinmonth[august]    := 31;
  daysinmonth[september] := 30;
  daysinmonth[october]   := 31;
```

```
  daysinmonth[november]  := 30;
  daysinmonth[december]  := 31;

  invert[1]      :=      january;
  invert[2]      :=      february;
  invert[3]      :=      march;
  invert[4]      :=      april;
  invert[5]      :=      may;
  invert[6]      :=      june;
  invert[7]      :=      july;
  invert[8]      :=      august;
  invert[9]      :=      september;
  invert[10]     :=      october;
  invert[11]     :=      november;
  invert[12]     :=      december;

END (* PROCEDURE initmonths *);

PROCEDURE datein(VAR thisdate: date);
VAR
  m,d: INTEGER;
BEGIN

  write('input date(mm dd)===>');
  readln(m,d);

  thisdate.mon := invert[m];
  thisdate.day := d;

END;

PROCEDURE control(cmd:INTEGER);
BEGIN

  write(lst,chr(cmd));

END;

BEGIN

  initmonths;
  rewrite(lst,'printer:');

  write('your name please===>');
  readln(name);

  write('input starting weight===>');
  readln(start);

  datein(dateday);

  write('input number of days to diet===>');
  readln(lastday);
```

LISTING 16.1 (cont.)

```
write('input calories per day===>');
readln(calperday);

control(enhance);
control(tiny);

writeln(lst);
write(lst,'  projected weight loss graph for: ');
writeln(lst,name);
writeln(lst);
write(lst,'initial weight: ');
writeln(lst,round(start):7);
write(lst,'calories per day: ');
writeln(lst,round(calperday):5);
writeln(lst);
writeln(lst);

control(normal);
control(small);

lowscale := round(start) - 45;
barweight[1] :=  start;

FOR thisday := 2 TO lastday DO
BEGIN

  dailyweight := barweight[thisday-1];

  deprivation := dailyweight*calsperlb - calperday;
  barweight[thisday] := dailyweight - (deprivation/poundoffat);

END;

write(lst,'  day                    weight (each * = 1 lb.)');
writeln(lst);
writeln(lst);
write(lst,'          ');
write(lst,lowscale:3);
FOR i := 1 TO 9 DO
BEGIN

  write(lst,'..!');
  next := lowscale + 5*i;
  write(lst,next:3);

END;
writeln(lst);
writeln(lst);

FOR thisday := 1 TO lastday DO
BEGIN
```

```
    stars := trunc(barweight[thisday]+0.99) - lowscale + 1;

    writedate(dateday);
    tomorrow(dateday);
    write(lst,'    ');

    i := 1;
    WHILE i < stars DO
    BEGIN

      write(lst,' ');
      i := i + 1;
      IF (i MOD 5) = 1
      THEN
        write(lst,'!')
      (* endif *);

    END (* DO *);
    writeln(lst,'*');

  END;

  control(enhance);
  control(tiny);

  writeln(lst);
  write(lst,'  expected final weight: ');
  writeln(lst,trunc(barweight[lastday]+0.99):4);
  write(lst,'  calories per day allowed in maintenance diet: ');
  maintain := round(barweight[lastday]*15.0);
  writeln(lst,maintain:5);

END.
```

LISTING 16.1 (*cont.*)

```
  PROJECTED WEIGHT LOSS GRAPH FOR: DICK VILE

INITIAL WEIGHT:      208
CALORIES PER DAY:   1500


  DAY                        WEIGHT (EACH * = 1 LB.)

          163..!168..!173..!178..!183..!188..!193..!198..!203..!208
NOV 27          !      !      !      !      !      !      !      !      !*
NOV 28          !      !      !      !      !      !      !      !      !*
NOV 29          !      !      !      !      !      !      !      !      !*
NOV 30          !      !      !      !      !      !      !      !     *
DEC  1          !      !      !      !      !      !      !      !     *
DEC  2          !      !      !      !      !      !      !      !    *
DEC  3          !      !      !      !      !      !      !      !    *
DEC  4          !      !      !      !      !      !      !      !   *
DEC  5          !      !      !      !      !      !      !      !   *
DEC  6          !      !      !      !      !      !      !      !  *
DEC  7          !      !      !      !      !      !      !      !  *
DEC  8          !      !      !      !      !      !      !      !  *
DEC  9          !      !      !      !      !      !      !      !*
DEC 10          !      !      !      !      !      !      !      !*
DEC 11          !      !      !      !      !      !      !     *
DEC 12          !      !      !      !      !      !      !     *
DEC 13          !      !      !      !      !      !      !    *
DEC 14          !      !      !      !      !      !      !    *
DEC 15          !      !      !      !      !      !      !   *
DEC 16          !      !      !      !      !      !      !   *
DEC 17          !      !      !      !      !      !      !   *
DEC 18          !      !      !      !      !      !      !  *
DEC 19          !      !      !      !      !      !      !  *
DEC 20          !      !      !      !      !      !      !*
DEC 21          !      !      !      !      !      !      !*
DEC 22          !      !      !      !      !      !     *
DEC 23          !      !      !      !      !      !     *
DEC 24          !      !      !      !      !      !     *
DEC 25          !      !      !      !      !      !    *
DEC 26          !      !      !      !      !      !    *
DEC 27          !      !      !      !      !      !   *
DEC 28          !      !      !      !      !      !   *
DEC 29          !      !      !      !      !      !   *
DEC 30          !      !      !      !      !      !  *
DEC 31          !      !      !      !      !      !  *
JAN  1          !      !      !      !      !      !*
JAN  2          !      !      !      !      !      !*
JAN  3          !      !      !      !      !      !*
JAN  4          !      !      !      !      !     *
JAN  5          !      !      !      !      !     *

  EXPECTED FINAL WEIGHT:  192
  CALORIES PER DAY ALLOWED IN MAINTENANCE DIET:  2870
```

LISTING 16.2 SAMPLE OUTPUT FROM DIET GRAPH PROGRAM

```
(*$s+*)
UNIT calendar;

INTERFACE

TYPE

  months  =  (january,february,march,april,
              may,june,july,august,september,
              october,november,december);

  date  =  RECORD

             mon:       months;
             day:       1..31;

           END;

VAR

  elapsed:       ARRAY[january..december] OF INTEGER;
  daysinmonth:   ARRAY[january..december] OF INTEGER;
  monthnames:    ARRAY[january..december] OF STRING[3];
  invert:        ARRAY[1..12] OF months;

  PROCEDURE writedate(VAR lst:text; thisdate: date);
  PROCEDURE tomorrow(VAR today:date);
  PROCEDURE datein(VAR thisdate:date);

IMPLEMENTATION

(*****************************************)
(*     w  r  i  t  e  d  a  t  e         *)
(*****************************************)

PROCEDURE writedate;
BEGIN

  write(lst,monthnames[thisdate.mon] );
  write(lst,thisdate.day:3);

END (* PROCEDURE writedate *);

(*****************************************)
(*   t   o   m   o   r   r   o   w     *)
(*****************************************)

PROCEDURE tomorrow;
VAR
  i:  INTEGER;
  d:  0..31;
  m:  months;
```

LISTING 16.3 CALENDAR ROUTINES UNIT

```
BEGIN

  m := today.mon;
  d := today.day;

  IF d+1 > daysinmonth[m]
  THEN
  BEGIN

    today.day := 1;

    IF today.mon = december
    THEN
      today.mon := january
    ELSE
      today.mon := succ(today.mon)
    (* endif *);

  END (* IF d+1> ... *)
  ELSE
    today.day := today.day + 1
  (* endif *);

END (* PROCEDURE tomorrow *);

(*****************************************)
(*    i  n  i  t  m  o  n  t  h  s     *)
(*****************************************)

PROCEDURE initmonths;
VAR
  i:  INTEGER;
  m:  months;
BEGIN

  monthnames[january]   := 'jan';
  monthnames[february]  := 'feb';
  monthnames[march]     := 'mar';
  monthnames[april]     := 'apr';
  monthnames[may]       := 'may';
  monthnames[june]      := 'jun';
  monthnames[july]      := 'jul';
  monthnames[august]    := 'aug';
  monthnames[september] := 'sep';
  monthnames[october]   := 'oct';
  monthnames[november]  := 'nov';
  monthnames[december]  := 'dec';

  daysinmonth[january]   := 31;
  daysinmonth[february]  := 28;
  daysinmonth[march]     := 31;
  daysinmonth[april]     := 30;
```

```
  daysinmonth[may]       := 31;
  daysinmonth[june]      := 30;
  daysinmonth[july]      := 31;
  daysinmonth[august]    := 31;
  daysinmonth[september] := 30;
  daysinmonth[october]   := 31;
  daysinmonth[november]  := 30;
  daysinmonth[december]  := 31;

  invert[1]      :=      january;
  invert[2]      :=      february;
  invert[3]      :=      march;
  invert[4]      :=      april;
  invert[5]      :=      may;
  invert[6]      :=      june;
  invert[7]      :=      july;
  invert[8]      :=      august;
  invert[9]      :=      september;
  invert[10]     :=      october;
  invert[11]     :=      november;
  invert[12]     :=      december;

END (* PROCEDURE initmonths *);

(*****************************************)
(*        d   a   t   e   i   n          *)
(*****************************************)

PROCEDURE datein;
VAR
  m,d: INTEGER;
BEGIN

  write('input date(mm dd)===>');
  readln(m,d);

  thisdate.mon := invert[m];
  thisdate.day := d;

END;


BEGIN

  initmonths;

END.
```

LISTING 16.4 DIET GRAPH WITH UNIT

```
(*****************************************)
(*                                       *)
(*     D  i  e  t  g  r  a  p  h         *)
(*                                       *)
(*     This program constructs a bar     *)
(* graph which represents the expect-    *)
(* weight loss during a caloric de-      *)
(* privation diet.  The loss is cal-     *)
(* culated based on the following        *)
(* assumptions:                          *)
(*                                       *)
(*   1. A pound of fat (body weight      *)
(*      lost first) is equivalent to     *)
(*      3500 calories.                   *)
(*   2. The body requires 15 calories    *)
(*      per pound to sustain a given     *)
(*      weight level.                    *)
(*   3. The dieter will adhere to a      *)
(*      regimen of so many calories      *)
(*      per day for the duration of      *)
(*      the projected dieting period.    *)
(*                                       *)
(*****************************************)

PROGRAM dietgraph;

(*$ucalendar.code*)
USES calendar;

CONST

  poundoffat      =        3500.0;
  calsperlb       =          15.0;
  enhance         =             1;
  normal          =             2;

  tiny            =            31;
  small           =            30;
  medium          =            29;
  large           =            28;

VAR

  lst:                     text;

  name:                    STRING;

  barweight:     ARRAY[1..365] OF REAL;

  start:                   REAL;
  calperday:               REAL;
  deprivation:             REAL;
```

```
  dailyweight:          REAL;
  loss:                 REAL;

  stars:                INTEGER;
  i:                    INTEGER;
  dateday:              date;
  thisday:              INTEGER;
  lastday:              INTEGER;
  lowscale:             INTEGER;
  next:                 INTEGER;
  maintain:             INTEGER;

PROCEDURE control(cmd:INTEGER);
BEGIN

  write(lst,chr(cmd));

END;

BEGIN

  rewrite(lst,'printer:');

  write('your name please===>');
  readln(name);

  write('input starting weight===>');
  readln(start);

  datein(dateday);

  write('input number of days to diet===>');
  readln(lastday);

  write('input calories per day===>');
  readln(calperday);

  control(enhance);
  control(tiny);

  writeln(lst);
  write(lst,'  projected weight loss graph for: ');
  writeln(lst,name);
  writeln(lst);
  write(lst,'initial weight: ');
  writeln(lst,round(start):7);
  write(lst,'calories per day: ');
  writeln(lst,round(calperday):5);
  writeln(lst);
  writeln(lst);

  control(normal);
  control(small);
```

LISTING 16.4 (*cont.*)

```
lowscale := round(start) - 45;
barweight[1] :=  start;

FOR thisday := 2 TO lastday DO
BEGIN

  dailyweight := barweight[thisday-1];

  deprivation := dailyweight*calsperlb - calperday;
  barweight[thisday] := dailyweight - (deprivation/poundoffat);

END;

write(lst,'  day                    weight (each * = 1 lb.)');
writeln(lst);
writeln(lst);
write(lst,'          ');
write(lst,lowscale:3);
FOR i := 1 TO 9 DO
BEGIN

  write(lst,'..!');
  next := lowscale + 5*i;
  write(lst,next:3);

END;
writeln(lst);
writeln(lst);

FOR thisday := 1 TO lastday DO
BEGIN

  stars := trunc(barweight[thisday]+0.99) - lowscale + 1;

  writedate(lst,dateday);
  tomorrow(dateday);
  write(lst,'   ');

  i := 1;
  WHILE i < stars DO
  BEGIN

    write(lst,' ');
    i := i + 1;
    IF (i MOD 5) = 1
    THEN
      write(lst,'!')
    (* endif *);

  END (* DO *);
  writeln(lst,'*');
```

```
  END;

  control(enhance);
  control(tiny);

  writeln(lst);
  write(lst,'  expected final weight: ');
  writeln(lst,trunc(barweight[lastday]+0.99):4);
  write(lst,'  calories per day allowed in maintenance diet: ');
  maintain := round(barweight[lastday]*15.0);
  writeln(lst,maintain:5);

END.
```

LISTING 16.5 LOW RESOLUTION GRAPHICS UNIT

```
(*$s+*)
UNIT lowresgr;

  INTERFACE

USES (*$u 5:peekpoke.code*) peekpoke;

CONST

  lores          =        -16298;
  graphics       =        -16304;
  mixtext        =        -16301;
  alltext        =        -16303;
  fullscreen     =        -16302;

  black          =             0;
  magenta        =             1;
  darkblue       =             2;
  purple         =             3;
  darkgreen      =             4;
  grey           =             5;
  medblue        =             6;
  lightblue      =             7;
  brown          =             8;
  orange         =             9;
  alsogrey       =            10;
  pink           =            11;
  green          =            12;
  yellow         =            13;
  aqua           =            14;
  white          =            15;

TYPE

  colorval       =        0..15;

VAR
```

LISTING 16.5 (*cont.*)

```
  color:        colorval;

  rowaddress:   ARRAY[0..23] OF INTEGER;

PROCEDURE initrowaddr;
PROCEDURE setcolor(c:colorval);
PROCEDURE plot(row: INTEGER; col: INTEGER);
PROCEDURE hline(col1,col2: INTEGER; row: INTEGER);
PROCEDURE vline(row1,row2: INTEGER; col: INTEGER);
PROCEDURE settext;
PROCEDURE gr;      (* note: sets full screen *)
FUNCTION  screen(row: INTEGER; col: INTEGER): colorval;
  IMPLEMENTATION

(****************************************)
(* i  n  i  t  r  o  w  a  d  d  r    *)
(****************************************)

PROCEDURE initrowaddr;
VAR
  i: INTEGER;
BEGIN

  rowaddress[0] := 1024;
  rowaddress[1] := 1152;
  rowaddress[2] := 1280;
  rowaddress[3] := 1408;
  rowaddress[4] := 1536;
  rowaddress[5] := 1664;
  rowaddress[6] := 1792;
  rowaddress[7] := 1920;
  rowaddress[8] := 1064;
  rowaddress[9] := 1192;
  rowaddress[10] := 1320;
  rowaddress[11] := 1448;
  rowaddress[12] := 1576;
  rowaddress[13] := 1704;
  rowaddress[14] := 1832;
  rowaddress[15] := 1960;
  rowaddress[16] := 1104;
  rowaddress[17] := 1232;
  rowaddress[18] := 1360;
  rowaddress[19] := 1488;
  rowaddress[20] := 1616;
  rowaddress[21] := 1744;
  rowaddress[22] := 1872;
  rowaddress[23] := 2000;

END (* PROCEDURE initrowaddr *);


(****************************************)
(*   s   e   t   c   o   l   o   r    *)
(****************************************)
```

```
PROCEDURE setcolor;
BEGIN

  color := c;

END (* PROCEDURE setcolor *);

(*****************************************)
(*      p     l     o     t              *)
(*****************************************)

PROCEDURE plot;
VAR
  hi,lo: 0..15;
  old: byte;
  spot: INTEGER;
BEGIN

  spot := rowaddress[row DIV 2] + col;

  old  := peek(spot);
  hi   := old DIV 16;
  lo   := old MOD 16;

  IF (row MOD 2) = 0
  THEN
    poke(spot,color+16*hi)
  ELSE
    poke(spot,lo+color*16)
  (* endif *);

END (* PROCEDURE plot *);

(*****************************************)
(*      h    l    i    n    e           *)
(*****************************************)

PROCEDURE hline;
VAR
  i: INTEGER;
BEGIN

  FOR i := col1 TO col2 DO
    plot(row,i)
  (* enddo *);

END (* PROCEDURE hline *);

(*****************************************)
(*      v    l    i    n    e           *)
(*****************************************)

PROCEDURE vline;
VAR
```

LISTING 16.5 *(cont.)*

```
  i:  INTEGER;
BEGIN

  FOR i := row1 TO row2 DO
    plot(i,col)
  (* enddo *);

END (* PROCEDURE vline *);

(*****************************************)
(*      s   e   t   t   e   x   t        *)
(*****************************************)

PROCEDURE settext;
BEGIN

  poke(alltext,0);

END (* PROCEDURE settext *);

(*****************************************)
(*             g    r                    *)
(*****************************************)

PROCEDURE gr;
VAR
  i: INTEGER;
BEGIN

  poke(lores,0);
  poke(fullscreen,0);
  poke(graphics,0);
  WITH memory DO
  BEGIN

    address := 1024;
    fillchar(pointer^[0],1024,chr(0));

  END (* WITH memory DO *);

END (* PROCEDURE gr *);

(*****************************************)
(*         s   c   r   e   e   n         *)
(*****************************************)

FUNCTION screen;
VAR
  hi,lo: 0..15;
  old: byte;
  spot: INTEGER;
BEGIN

  spot := rowaddress[row DIV 2] + col;
```

```
    old  := peek(spot);
    hi   := old DIV 16;
    lo   := old MOD 16;

    IF (row MOD 2) = 0
    THEN
      screen := hi
      ELSE
        screen := lo
      (* endif *);

    END (* FUNCTION screen *);

    BEGIN

      initrowaddr;

    END.
```

LISTING 16.6 PEEK AND POKE UNIT

```
(*$s+*)
UNIT peekpoke;

  INTERFACE

TYPE

  byte          =       0..255;
  memloc        = PACKED ARRAY[0..1] OF byte;

  access        = RECORD CASE BOOLEAN OF

                    false: (address: INTEGER);
                    true:  (pointer: ^memloc);

                  END;

VAR

  memory:       access;

PROCEDURE poke(spot: INTEGER; value: byte);
FUNCTION  peek(spot: INTEGER): byte;

  IMPLEMENTATION


(*****************************************)
(*          p   o   k   e                *)
(*****************************************)
```

LISTING 16.6 *(cont.)*

```
PROCEDURE poke;
BEGIN

  WITH memory DO
  BEGIN

    address      := spot;
    pointer^[0] := value;

  END (* WITH memory DO *);

END (* PROCEDURE poke *);

(*****************************************)
(*            p   e   e   k              *)
(*****************************************)

FUNCTION peek;
BEGIN

  WITH memory DO
  BEGIN
    address := spot;
    peek    := pointer^[0];

  END (* WITH memory DO *);

END (* FUNCTION peek *);

BEGIN
END (* UNIT peekpoke *).
```

LISTING 16.7 HOUSE PICTURE DEMO

```
PROGRAM house;

USES (*$u 5:peekpoke.code*) peekpoke,
     (*$u 5:lowres.code*) lowresgr;

TYPE

  direction      =     (up,down);

VAR

  location:      INTEGER;

  row,
  col:           INTEGER;

  ch:            CHAR;
```

```
(*****************************************)
(*           b   l   o   b               *)
(*****************************************)

PROCEDURE blob(c1,c2,r1,r2:INTEGER);
VAR
  i:     INTEGER;
BEGIN

  FOR i := r1 TO r2 DO
    hline(col+c1,col+c2,row+i)
  (* enddo *);

END (* PROCEDURE blob *);

(*****************************************)
(*           b   o   x                   *)
(*****************************************)

PROCEDURE box(c1,c2,r1,r2:INTEGER);
BEGIN

  hline(col+c1,col+c2,row+r1);
  hline(col+c1,col+c2,row+r2);
  vline(row+r1,row+r2,col+c1);
  vline(row+r1,row+r2,col+c2);

END (* PROCEDURE box *);

(*****************************************)
(*         s  t  a  i  r  s              *)
(*****************************************)

PROCEDURE stairs(d:direction; c1,c2,r1,l:INTEGER);

VAR
  i:     INTEGER;
BEGIN

  CASE d OF

    up:     FOR i := 0 TO l-1 DO
              hline(col+c1-i,col+c2,row+r1+i);
    down:   FOR i := 0 TO l-1 DO
              hline(col+c1,col+c2+i,row+r1+i);

  END (* CASE d OF *);

END (* PROCEDURE stairs *);

(*****************************************)
(*     d  i  a  g  o  n  a  l            *)
(*****************************************)
```

LISTING 16.7 *(cont.)*

```
PROCEDURE diagonal(d:direction; r,c,l:INTEGER);
VAR
  i:     INTEGER;
BEGIN

  CASE d OF

    up:     FOR i := 0 TO l-1 DO
              plot(r-i,c-i);

    down:   FOR i := 0 TO l-1 DO
              plot(r+i,c+i);

  END (* CASE d OF *);

END (* PROCEDURE diagonal *);

BEGIN

  gotoxy(0,21);
  row := 0;
  col := 0;
  gr;

  setcolor(white);
  blob(0,10,0,10);
  blob(20,39,0,20);
  hline(9,20,0);
  setcolor(darkblue);
  blob(10,19,1,9);
  blob(1,29,10,20);
  plot(1,20);
  plot(2,9);
  plot(11,30);
  stairs(up,8,9,3,7);
  stairs(down,20,21,2,8);
  stairs(down,30,31,12,8);
  setcolor(orange);
  diagonal(up,10,0,10);
  diagonal(down,1,21,19);
  vline(10,29,0);
  vline(20,38,39);
  box(12,18,2,8);
  box(3,9,10,18);
  box(12,18,10,18);
  box(21,27,10,18);
  box(3,18,20,28);
  hline(20,29,20);
  vline(20,29,20);
  vline(20,29,29);
  hline(20,39,30);
  hline(21,39,31);
  hline(22,39,32);
```

```
    hline(23,39,33);
    hline(35,38,34);
    hline(36,38,35);
    hline(37,38,36);
    plot(37,38);
    setcolor(green);
    diagonal(down,30,30,10);
    stairs(down,0,19,30,10);
    setcolor(grey);
    hline(25,33,34);
    hline(26,34,35);
    hline(27,35,36);
    hline(28,36,37);
    hline(29,37,38);
    hline(30,38,39);
    blob(31,37,21,9);
    setcolor(pink);
    blob(4,17,21,7);
    blob(21,28,21,9);
    blob(13,17,3,5);
    blob(4,8,11,7);
    blob(13,17,11,7);
    blob(22,26,11,7);
    setcolor(brown);
    hline(13,17,5);
    vline(3,7,15);
    hline(4,8,14);
    hline(13,17,14);
    hline(22,26,14);
    vline(11,17,6);
    vline(11,17,15);
    vline(11,17,24);
    setcolor(grey);
    blob(31,37,21,9);
    hline(4,17,22);
    hline(4,17,25);
    vline(21,27,5);
    vline(21,27,8);
    vline(21,27,12);
    vline(21,27,15);
    setcolor(brown);
    blob(23,26,22,8);
    vline(35,38,4);
    setcolor(purple);
    blob(3,5,33,2);

    readln(ch);
    settext;
    write(chr(12));

END.
```

Chapter

17

APPLE Assembler: Features and Use

To use a well-worn phrase, we now get to the core of the APPLE; namely, the 6502 microprocessor. The 6502 CPU may be programmed directly in binary or hexadecimal machine language or in symbolic assembly language. While it is necessary to have some understanding of the machine language, we shall concentrate on programming in assembly language. As in other sections of the book, we begin with a brief overview of the language and its highlights.

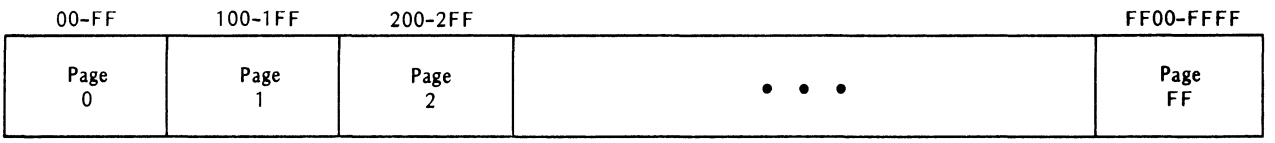
1. INTRODUCTION

We do not have the space for a complete course of study for beginning 6502 assembly-language programmers. That is not our aim. We will assume that the interested reader has some familiarity with the subject already. Perhaps you have read one of the many available books on 6502 assembly-language programming. Or perhaps you have worked your way through some of the articles in personal computing magazines which use machine code in programs. In either case, you are not totally intimidated by the low-level concepts of the APPLE II machine.

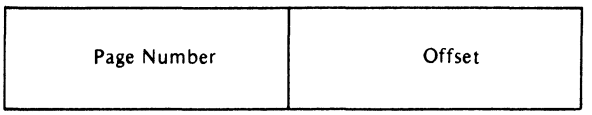
Since reading articles which contain samples of 6502 code is a good way to extend your knowledge, we have included at the end of this chapter a selected bibliography of interesting material.

The emphasis in this section will not be on traditional applications of assembly language, such as controlling peripheral hardware devices, or writing disk operating systems. Rather, we shall be concentrating on areas which may be more directly approachable by the casual user: service routines or utilities callable from BASIC, graphics routines, screen manipulation, and so on. Much of the code presented could easily be implemented in BASIC, but if done so would be unbearably slow. Other code depends on the peculiarities of the APPLE II firmware and would be much more difficult to implement in BASIC.

All the sample programs presented have been prepared using the APPLE Assembler/Editor which is part of the DOS Tool Kit marketed by Apple Computer. This is not by any means the only commercially available APPLE assembler, but is perhaps the most standard one we could have chosen. If you already own another assembler, it may be necessary for you to convert some of the source code into the syntax acceptable by your assembler.



6502 RAM: 256 Pages of 256 Locations = 65536 Locations



6502 Address

FIGURE 17.1 6502 Memory Addressing and Layout

2. THE 6502 CPU

In order to program in 6502 assembler, familiarity with the features of the 6502 CPU is a must. We give a brief review of the basics here.

The 6502 is an 8-bit microprocessor. This means that it *operates* on data eight bits (one byte) at a time. Most 6502 operations expect operands which are one byte in size.

The 6502 CPU can accommodate up to 64 × 1024 = 65536 bytes of random access memory (RAM). Each byte in memory is assigned a unique number between 0 and 65535, which is called its *address*.

It is often convenient to think of 6502 memory in terms of *pages*. A *page* is a contiguous chunk of 256 memory locations, which starts at an address which is a multiple of 256. The 6502 has up to 256 pages of memory, each of which contains 256 bytes:

$$256 \times 256 = 65536$$

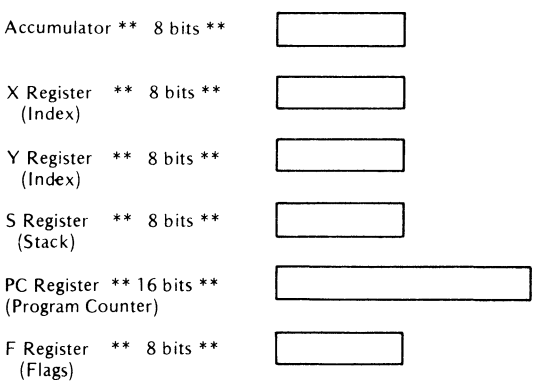
The address of any memory location may be specified by giving its page number and its offset with the page. Since values of 0 to 255 can be represented in eight bits, this means that both the page number and the offset require eight bits (and no more) to store. When the two are concatenated, they form a 16-bit number. When all this is expressed in hexadecimal notation, the address turns out to be a four-digit hex number. The high-order two digits tell us the page and the low-order two digits tell us the offset in the page. Figure 17.1 illustrates these concepts.

The 6502 CPU contains several registers which are used for data storage, manipulation, and movement to and from RAM. Figure 17.2 shows the 6502 register set.

The accumulator register holds the results of arithmetic and logical operations. The X and Y registers serve as data storage registers. They are also used in the fancier 6502 "addressing modes." An *addressing mode* is a method by which an address in RAM may be specified in a machine instruction. We deal with 6502 addressing modes below.

The stack pointer register indicates the top of the processor stack, which is used for storing return addresses from machine-language subroutines, as well as for temporary data storage. The Program Counter or PC register contains the address of the next 6502 instruction to be executed. The contents of the Processor Flags register are treated as eight individual bits, each of which gives information on a particular condition within the CPU. For example, the Z (for *zero*) flag will be equal to 1 if the result of the last arithmetic operation was zero.

FIGURE 17.2 6502 Processor Registers



Opcodes

The CPU carries out its work by executing individual *operations*. In the machine, each operation is indicated by a unique 8-bit number, called its operation code, or *opcode* for short. In assembly language, opcodes are represented by short mnemonics, usually three or four letters long. The assembler program is responsible for translating the opcode mnemonics into their correct numeric form.

For an exhaustive description of what each 6502 opcode can do, refer to your assembly-language textbook.

Opcodes may be categorized by their function. The division of 6502 opcodes into categories is somewhat arbitrary, but here is one possibility:

- Arithmetic opcodes: i.e., opcodes that modify the contents of the accumulator, X or Y registers, or some memory location.

ADC	**AND**	**ASL**	**EOR**
LSR	**ROL**	**ROR**	**SBC**
DEX	**DEY**	**INX**	**INY**
DEC			

- Data Transfer opcodes: opcodes that move data from memory to registers or back, or from one register to another.

LDA	**STA**	**LDX**	**STX**
LDY	**STY**	**PHA**	**PLA**
TAX	**TXA**	**TAY**	**TYA**
TXS	**TSX**		

- Branching opcodes: opcodes that cause the CPU to execute instructions out of sequential order by affecting the contents of the PC register.

BCC	**BCS**	**BEQ**	**BNE**
BMI	**BPL**	**BVC**	**BVS**
JMP	**JSR**	**RTS**	**RTI**

- Opcodes which affect the flags: opcodes that explicitly or implicitly change the value of a processor flag. (Note the overlap with some of the arithmetic opcodes.)

ADC	**ASL**	**BIT**	**CLC**
CLD	**CLI**	**CLV**	**CMP**
CPX	**CPY**	**DEC**	**DEX**
DEY	**LDA**	**LDX**	**LDY**
LSR	**PHP**	**PLP**	**ROL**
ROR	**SBC**	**SEC**	**SED**
SEI	**TXA**	**TAX**	**TAY**
TYA	**TSX**	**INC**	**INX**
INY			

Processor Flags

The Processor Flags register contains eight bits. Seven of those bits are used by the 6502 to remember certain conditions within the machine.

Flags are the heart and soul of microprocessor programming. For those of you who may be new to the subject, let us define our terms: A *flag* is a brightly colored piece of cloth that hangs from a flagpole. But seriously, now, a *processor flag* is an indicator in the CPU of a computer system. The indicator may either be either ON or OFF. Its state may be changed by certain machine instructions and tested or sensed by other machine instructions. There are seven flags available to the user of the 6502 CPU:

Flag Name	*Abbreviation*
Negative	N
Overflow	V
Break command	B
Decimal mode	D
Interrupt disable	I
Zero	Z
Carry	C

The programs we present primarily use the zero, carry, and negative flags. When studying the programs, you should pay careful attention to their use.

Addressing Modes

The 6502 provides a rich variety of ways to get at memory. These so-called addressing modes give the programmer flexibility, especially in handling data structures in memory. The modes available are as follows:

Immediate addressing
Absolute addressing (also called Direct addressing)
Zero page addressing
Accumulator addressing
Zero page indexed addressing
Absolute indexed addressing
Indexed indirect addressing
Indirect indexed addressing

Each of these modes has its own special uses. Especially important are the zero page addressing modes. Page zero consists of the first 256 RAM memory locations, those with addresses 0 to 255. Zero page addressing modes do not require the use of the page number when specifying a memory address. The page number of zero is *implied* when these modes are used. This gives page zero a very special place in the 6502 hall of fame.

Instructions which use zero page addressing require only two bytes of memory to store: one byte for the opcode and one byte for the zero page offset—again, the page number of 0 is implied. Instructions which use absolute addressing modes require three bytes of memory to store: one byte for the opcode, and two bytes for the address—one for the page number and one for the offset in the page.

Now assembly language programmers are a penurious lot. They want to squeeze their code into as small a space as possible. Therefore, zero page addressing modes are quite favored by these folks. In turn, the memory in page zero tends to get preempted for storing the data which may be referred to by economical two-byte zero page mode instructions. This is the reason for you to be aware of zero page usage of programs with which you must interact—such as Integer or APPLESOFT BASIC or APPLE DOS. If you decide to use a page zero location which is already reserved, for example, by DOS, you could easily clobber important information which DOS needs in order to remain healthy.

3. ASSEMBLY LANGUAGE

Assembly programs have their own unique requirements and flavor. In this section, we review some of that flavor. For detailed descriptions of the APPLE Assembler, you should consult the manual *APPLE 6502 Assembler/Editor* that comes with the DOS Tool Kit.

Source Line Format

Assembly instructions are much more regular than Pascal or BASIC statements. They consist of "fields," each of which serves a specific well-defined purpose. Figure 17.3 shows the format of APPLE assembler statements.

LABEL OPCODE OPERAND ;COMMENT

FIGURE 17.3 APPLE Assembler Source Statement Format

Briefly, here is the function of each field:

• LABEL This gives a "name" to the statement. The label actually represents an address. It is equated to the address in memory where the machine code for the source line is eventually assembled.

• OPCODE This field has a dual nature. It may contain a mnemonic that stands for a machine instruction, such as LDA, STA, INX, ROR, etc., or it may contain a mnemonic which stands for an assembler instruction; a so-called *pseudo-op*. Pseudo-ops represent instructions to the assembler, footnotes, if you will, containing extra information about how to translate the assembly code into machine code. We discuss this concept in more detail below.

• OPERAND This field also serves a dual role. If the opcode field of a given statement contains a machine opcode, then the operand field contains information about the data which the opcode will operate upon. This information is expressed using one of the 6502 addressing modes indicated above. Figure 17.4 shows the syntax used for each addressing mode. If the opcode field of a given statement contains an assembler pseudo-op (also sometimes called a *directive*), then the operand field will contain any extra information needed by the pseudo-op.

• COMMENT The comment field needs little comment. It is provided for the programmer's use. It should contain informative and elucidating commentary on the *meaning* of the statement to which it applies.

A comment on COMMENTS

In assembly code, it is a temptation to put in comments that merely paraphrase the opcode of the instruction. While this cannot always be avoided, it is preferable that

Addressing Mode	*Syntax*	*Example Instructions*
Immediate	#value	LDA #$00 CMP #'A
Absolute	label or constant	LDA NUMBER JMP $FDED
Zero Page	label or constant (value < 256)	CMP WNDLFT LDA CH STA $30
Zero Page Indexed	label,X or label,Y or constant,X or constant,Y	STA WNDLFT,X LDA $F0,X STA $30,Y
Absolute Indexed	label,Y or constant,X or constant,Y	LDA TICHARS,X STA $800,Y

Indexed Indirect	(label,X) label must be zero page—or (constant,X) constant must be < 256	LDA (A1L,X)
		STA (CHRTAB,X)
		LDA ($48,X)
		STA ($E0,X)

Indirect Indexed	(label),Y or (constant),Y label or constant must be zero page	LDA (BASL),Y
		STA (BAS2L),Y
		LDA ($28),Y
		STA ($2A),Y

FIGURE 17.4 6502 Addressing Modes

```
INCPGNUM  INC  PGNUM+1     ;BUMP UNITS DIGIT
          LDA  PGNUM+1     ;CHECK IT
          CMP  #$39        ;IS IT > 9?
          BCS  PGRTS       ;NO — GO BACK
          LDA  #$30        ; YES     CHANGE TO 0
          STA  PGNUM+1     ;SAVE IT
          LDA  PGNUM       ;NOW TENS DIGIT
          CMP  #$20        ; IF = " ",THEN MAKE IT 1
          BEQ  FIRST
          INC  PGNUM       ;OTHERWISE — BUMP IT
          RTS              ;AND QUIT
FIRST     LDA  #$31        ;MAKE IT A 1
          STA  PGNUM       ;SAVE IN TENS DIGIT
PGRTS     RTS
```

(a)

```
INCPGNUM  INC  PGNUM+1     ;INCREMENT PGNUM+1
          LDA  PGNUM+1     ;LOAD IT INTO AC
          CMP  #$39        ;COMPARE TO 39 HEX
          BCS  PGRTS       ;BRANCH TO PGRTS IF CARRY SET
          LDA  #$30        ;PUT 30 HEX INTO AC
          STA  PGNUM+1     ;STORE AC INTO PGNUM+1
          LDA  PGNUM       ;NOW GET PGNUM
          CMP  #$20        ;COMPARE IT TO 20 HEX
          BEQ  FIRST       ;BRANCH TO LABEL FIRST
          INC  PGNUM       ;ADD ONE TO PGNUM
FIRST     LDA  #$31        ;PUT 31 HEX INTO AC
          STA  PGNUM       ;AND STORE INTO PGNUM
PGRTS     RTS              ;RETURN FROM SUBROUTINE
```

(b)

FIGURE 17.5 Useful and Worthless COMMENTS

comments amplify upon the *meaning* of the instruction, not just put the instruction into different words. Figure 17.5 gives two versions of an assembly code fragment. The first version has useful comments; the second version may as well have no comments at all.

4. THE ROLE OF THE ASSEMBLER

The assembler has the task of translating a program from source assembly language to target machine language. At first blush this may seem to be easy. Just turn mnemonic opcodes into numeric values and symbolic labels into machine addresses. It's almost that easy, but not quite. Frequently, the programmer wishes to communicate extra information to the assembler. Such information tells the assembler how to do its job. The mechanism for this is the *assembler directive* or *pseudo-op*. A few examples are in order to give some idea of the possibilities.

Generating Data

Most assembly programs need to store numeric values to be used by the code of the program. In order to generate these values into the translated program, it is necessary to have certain pseudo-ops.

DFB—Define Byte

This directive is used to generate a single byte of data in the assembled program. For example, a typical assembly language program might contain statements such as:

```
ONE      DFB  1
FFHEX    DFB  $FF
MAXLIN   DFB  55
CHPLINE  DFB  80
```

and so on.

DW—Define Word

This directive is used to generate 16-bits quantities, for example 6502 addresses, into a program. This might be used to store a table of addresses of messages to be printed or a table of routines to be jumped to. Many other applications are possible as well.

Formatting the Listing

Most assemblers provide directives to assist the programmer in achieving a neat listing.

PAGE

This tells the assembler to start a new page in the listing. It allows the programmer to group the listing of various subroutines. A new group of subroutines may start on a new page. This makes reading the listing a lot simpler.

SKP N

This tells the assembler to skip N lines on the current listing page before printing the next line. This allows the programmer to insert blank lines in the listing. Again, this vertical formatting capability can lead to listings that are more readable and easy to use. This pseudo-op is required by the Tool Kit Assembler in order to achieve blank lines in the listing. Putting blank lines in the source causes syntax errors. Other assemblers are "smart" enough to ignore blank lines in the source. This obviates the necessity for a SKP directive in such assemblers.

LST ON/OFF

This directive allows certain sections of the listing to be suppressed. This is useful in longer assemblies when large portions of the program will remain unchanged from one version to the next. Rather than generate a new copy of the unchanged code each time, the programmer can tell the asembler to turn off the listing. This saves wear on the printer, time for the programmer, life for some trees, etc.

Translation of ASCII Strings

The way that the APPLE computer stores characters is somewhat nonstandard. It uses only 6 of 8 bits to tell it what the actual character is and uses the other two bits to determine the display mode. When it is necessary to generate data in standard ASCII format, we may want to defeat the use of these mode bits. The DOS Tool Kit Assembler gives us a pseudo-op to assist us: the MSB pseudo-op. MSB is short for "Most Significant Bit." The MSB directive has two forms:

MSB ON and **MSB OFF**

The first of these tells the assembler to turn the MSB of each character assembled ON. The second tells it to turn the MSB OFF. To see the practical effect of this, consider the example:

```
MSG     ASC /WELCOME TO APPLE TRIVIA/
```

If this line is assembled with MSB ON in effect, the result will be as follows:

```
0000D7  C5 CC   1 MSG   ASC/WELCOME
                        TO APPLE TRIVIA/
0003:C3 CF CD
0006:C5 A0 D4
0009:CF AO C1
000C:D0 D0 CC
000F:C5 A0 D4
0012:D2 C9 D6
0015:C9 C1
0017:
```

On the other hand, if it is assembled with MSB OFF in effect, the result would be:

```
0000:57 45 4C   1 MSG   ASC /WELCOME
                        TO APPLE TRIVIA/
0003:43 4F 4D
0006:45 20 54
0009:4F 20 41
000C:50 50 4C
000F:45 20 54
0012:52 49 56
0015:49 41
0017:
```

The second version is needed when the data must be in standard ASCII format. This is the case when interacting with APPLESOFT. The first version is used when producing messages that will be displayed in normal mode on the APPLE display.

Summary

The particular pseudo-ops made available vary widely from one assembler to another. You will have more difficulty with pseudo-ops than with instructions when converting programs. The degree of difference is unfortunately broader than similar differences between versions of BASIC or Pascal.

4. SELECTED BIBLIOGRAPHY OF 6502 ARTICLES

BISHOP, ROBERT J., "APPLE Kaleidoscope," *BYTE*, Vol. 4, Number 7, July 1979.

WOZNIAK, STEPHEN, "SWEET16: The 6502 Dream Machine," *BYTE*, Vol. 2, Number 11, Nov. 1977.

CROSSLEY, JOHN, "APPLESOFT Internal Entry Points," *The APPLE Orchard*, Vol. 1, Number 1, Mar/Apr 1980.

HYDE, RANDALL, "Converting INTEGER BASIC Programs to Assembly Language," *The APPLE Orchard*, Vol. 1, Number 1, Mar/Apr 1980.

McVAY, RAY, "Game Sounds," *CALL A.P.P.L.E.*, Sept. 1980.

REYNOLDS, WILLIAM III, "APPLESOFT Program Splitter," *CALL A.P.P.L.E.*, June 1980.

MOTTOLA, R.M., "Amper-Interpreter," *NIBBLE*, No. 6, 1980.

CROSSMAN, CRAIG, "Fun with APPLE's Assembler," *NIBBLE*, No. 5, 1980.

SUITOR, RICHARD F., "Applayer Music Interpreter," *The Best of Micro*.

WILLIAMS, RICHARD, "How to Use the Hooks," *MICRO*, 30, Nov. 1980.

MORRIS, GARY A., "Print USING for APPLESOFT," *MICRO*, 27, Oct. 1980.

VILE, RICHARD C., Jr., "Integer FLASH for the APPLE," *MICRO*, 37, June 1981.

COCHARD, STEVE, "APPLESOFT Error Messages from Machine Language," *MICRO*, 39, August 1981.

Chapter

18

Assembly-Language Techniques

In this chapter we present a potpourri of techniques of use in programming the APPLE II in assembly language. Some of these techniques are generally applicable in any 6502-based computer. Some depend on the particular structure of the APPLE II hardware, firmware, and software. We will just barely scratch the surface here, and the reader is advised to study the sample programs in the remainder of the book for similar examples.

1. MORE ON CARRY AND BORROW

Remember grade school arithmetic? "Carry the one," "borrow from the next place up," and so on. Little did you suspect that one day it would come back to haunt you during your programming career. Figure 18.1 gives a quick review of the use of the terms "carry" and "borrow" in arithmetic.

In the 6502 the carry flag functions as *both* carry and borrow indicator. There are many ways that a "carry" may occur (i.e., ways that the carry flag may be set). We consider some of the arithmetic operations that can cause this.

A carry will occur when the result of an addition is too large to fit into the accumulator. The 6502 accumulator holds an 8-bit number. The result of an addition must be between 0 and 255. Any addition which results in a number larger than 255 will *set* the carry flag.

Now comes the surprising part. You would expect the carry to be set if the result of a subtraction was a number less than 0. Such a situation is the logical equivalent of a grade school borrow. Sorry, guess again. If a 6502 subtraction results in a number less than zero, the carry flag is *cleared*. If the subtraction yields zero or a positive result, the carry is *set*. Since the first situation is the one we think of as a "borrow," the borrow is the inverse or opposite of the carry. Put in more mathematical terms:

- The borrow is the *complement* of the carry.

If the carry flag is set as the result of a subtraction, then there was no borrow. If the carry flag is cleared, then there was a borrow. Confused? I certainly am—every time I go over this stuff again.

The ADC and SBC Opcodes

The 6502 has just one instruction for addition and one instruction for subtraction. They are:

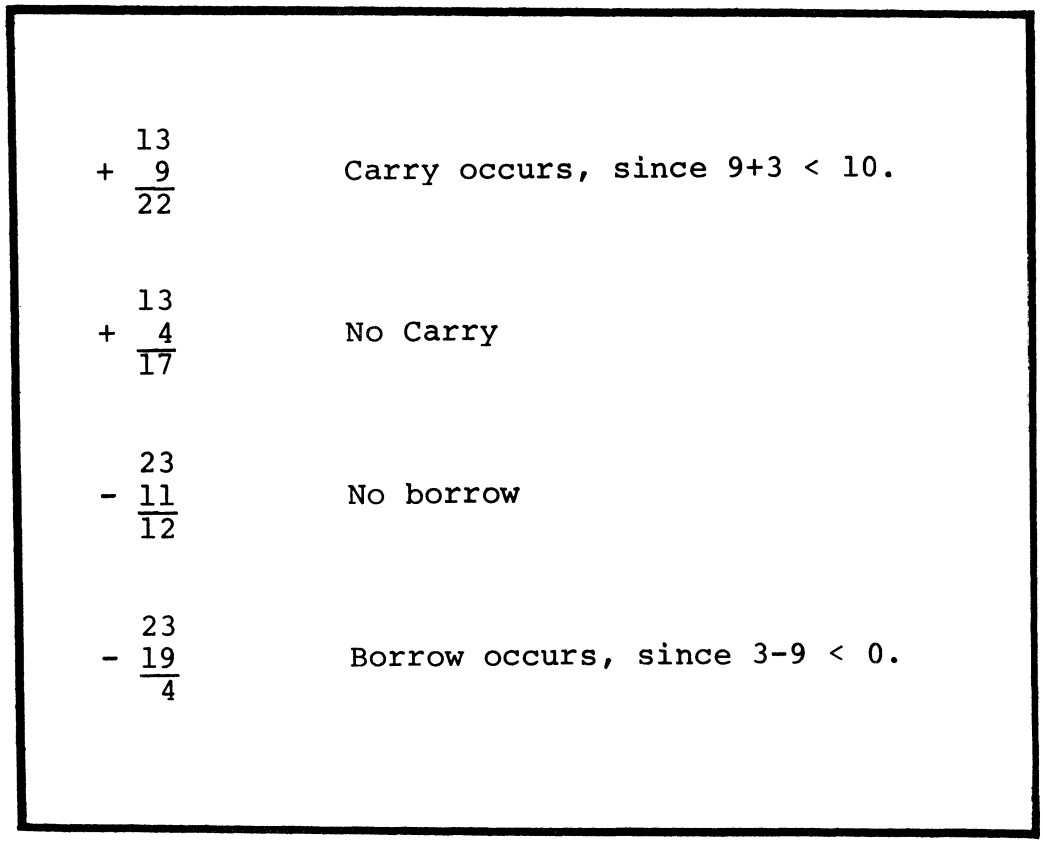


FIGURE 18.1 Grade School Carry and Borrow

ADC Add the contents of the accumulator to the contents of memory *with carry*.

SBC Subtract the contents of memory *with borrow* from the contents of the accumulator.

The carry flag is accounted for by *every* addition and subtraction that you perform on the 6502. Each addition adds in the carry flag and each subtraction subtracts out the borrow, or the complement of the carry. This can lead the unsuspecting into strange traps. Suppose you write the instruction:

```
SBC $0A     ;*** SUBTRACT 10
```

in your program. You may think that this will always result in 10 being subtracted from the contents of the accumulator. This is not true! If the carry happens to be clear, then the instruction above will cause 11, not 10, to be subtracted from the accumulator. Why? Just apply the definition faithfully:

SBC $0A Subtract the contents of memory with borrow: in this case the contents of memory specified is the immediate value of hex 0A, or decimal 10.

With borrow means that the complement of the carry flag will be subtracted along with the other value. If the carry is clear (=0), then its complement (the borrow) is set (=1) and the subtraction gives one smaller than indicated.

When in doubt set or clear the carry

Whenever you perform a subtraction, and you are not *absolutely sure* of the state of the carry flag, you should explicitly set it beforehand. This will guarantee that the borrow is clear.

Likewise, you should clear the carry before any ADC instruction for which you are not absolutely certain of the state of the carry. Unless, of course, you *want* the carry to be included in the addition.

The following short examples give a summary of the use of ADC and SBC in connection with the carry flag.

Instructions	*Comments*
ADC $01	Could add 1 or 2 to the contents of the accumulator carry set → add +2 carry clear → add +1

```
SEC
ADC $01      Will always add +2
CLC
ADC $01      Will always add +1
SBC $01      Could subtract 1 or 2 from the contents
               of the accumulator
                 carry set → subtract +1
                 carry clear → subtract +2
SEC
SBC $01      Will always subtract +1
CLC
SBC $01      Will always subtract +2
```

2. SIX TECHNIQUES THAT USE THE CARRY FLAG

In this section, we discuss six assembly language coding techniques which utilize the carry flag. We couch the discussion in terms of six APPLE Programming Tips.

Branching decisions based on comparisons

Recalling our discussion above, we know that the borrow flag is defined as the complement of the carry. Suppose P and Q are the names assigned to two memory locations in APPLE II RAM and consider the sequence of instructions:

```
LDA P
CMP Q
```

which causes the numbers in P and Q to be *compared*. Just how do we get our hands on the results of the comparison? We use the carry flag. The CMP opcode performs an implicit (fancy way of saying "behind your back") subtraction of the value of its argument from the contents of the accumulator. In this example, this would be written as P − Q. Now a borrow occurs in this situation if and only if Q is greater than P. Therefore a carry does not occur if and only if Q is greater than P. Finally, a carry *does* occur if and only if Q is not greater than P. That is, if and only if Q is less than or equal to P. So what, you say? All of this palaver stated simply means that the "BASIC-style" IF statement:

```
IF Q <= P THEN blah-blah-blah
```

may be translated into 6502 machine language as follows:

```
        LDA P
        CMP Q
        BCC DONT
          blah
          ...
          blah
          ...
          blah
DONT —
```

that is, the sequence of 6502 instructions represented by blah ... blah ... blah in the figure will be carried out if and only if Q <= P.

Testing a numeric quantity for an even or odd value

A number expressed in binary is even if and only if its least significant bit is a 0. Putting it another way, a number is odd if and only if its least significant bit is a 1.

If a program needs to determine whether a number is even or odd, this can be accomplished using the carry flag. The instruction:

```
ROR A
```

causes the least significant bit of the number stored in the accumulator to be "rotated" into the carry bit. Thus, if we are interested in the evenness of the value stored in P, we could write:

```
        LDA P
        ROR A
        BCC ITSEVEN
        ...        ;DO THIS IF IT's ODD
ITSEVEN —
```

Testing for a negative byte value

When a byte value is considered as a two's complement number (if you are not familiar with two's complement concepts, just sort of snooze through this example), it will be negative precisely when the *most* significant bit is a 1. The instruction:

```
ROL A
```

causes this bit to be placed into the carry flag. The sequence of instructions:

```
        LDA P
        ROL A
        BCC NONNEG
        ...        ;DO THIS IF NEGATIVE
NONNEG —
```

may be used to check for negative or non-negative values.

Unconditional relative branching

Absolute jump instructions in a program force the program always to be loaded into the same memory locations. Using relative branches instead of absolute branches will allow a program to be *position independent* or *self-relocating*. The problem is that the 6502 does not possess an unconditional relative branch instruction. Luckily, we can "fake" such an instruction by using the carry flag. Try either of the following:

```
SEC
BCS THERE
```

or

```
CLC
BCC THERE
```

Sixteen-bit shifts

Suppose a 16-bit quantity is stored in memory at locations P and P+1, with the least significant byte at P. The following sequence of instructions will cause this quantity to be shifted left one bit. (Note: this is equivalent to multiplication by 2.)

```
CLC
ROL P
ROL P+1
```

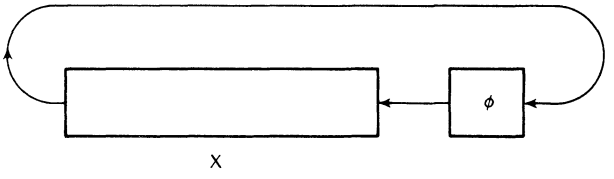
The first instruction, CLC, guarantees that the carry flag will contain a 0. The second instruction is pictured in Figure 18.2. Three things take place:

- The carry is placed in the least significant bit of P.
- The byte in P is shifted left one bit.
- The high bit of P is placed in the carry flag.

The next instruction is ROL P+1, which in turn causes:

- The carry flag, which contains the high bit of P because of the previous instruction, is placed in the least significant bit of P+1.

FIGURE 18.2 Rotate Instruction



- The byte in P+1 is shifted left one bit.
- The most significant bit of P+1 is placed in the carry flag.

The combined effect of all this is to shift the 16-bit number in P and P+1 left one bit.

Manufacturing counting loops

We can use the carry flag to build sequences of instructions which behave like BASIC FOR statements. For example, suppose we want to translate the statements:

```
10 FOR I = 1 TO 100
  ...
99 NEXT I
```

into 6502 machine language. First, we realize that the above may be rewritten as follows:

```
 9 I=1
10 ...

99 I = I+1
100 IF I <= 100 THEN 10
```

From there and our discussion of branching decisions, we arrive at the translation:

```
      LDA $01
      STA I
LOOP  ...

      INC I
      LDA $64
      CMP I
      BCS LOOP
```

3. USING 6502 ADDRESSING MODES

The 6502 has a large number of different addressing modes. We now discuss some of them and give short- or medium-length examples of their application in actual code.

Immediate Addressing

The immediate addressing mode allows constant argument values of up to 255 to be stored as part of an instruction. This avoids the tedium of always having to declare memory locations in which to store common con-

stants. Compare for example the following two translations of the BASIC Assignment statement I = 1:

Without Immediate Addressing	With Immediate Addressing
LDA ONE	LDA $01
STA I	STA I
...	
ONE DB 1	

Immediate addressing may also be used with word quantities as well. The trick is to split the word-sized constant into two byte-sized constants first. The following code fragment causes the address $FDED to be pushed onto the 6502 processor stack.

```
LDA $ED
PHA
LDA $FD
PHA
```

Page Zero Addressing and Direct Addressing

Direct memory addressing is analogous to direct referencing of a simple variable in a BASIC program. The assembly-language program must declare or reserve storage for each variable it wishes to address in such a fashion. This is done by using data storage pseudo-ops as explained in Chapter 17. The labels created by the use of such pseudo-ops may then be referenced by instructions using direct memory addressing. Thus, the use of variables in assembler is a two-stage process:

1. Create a name for the variable (label) and reserve space for it using a memory allocation pseudo-op. This step is analogous in many ways to the var declarations required in every Pascal program.
2. Refer to the label created via step 1 in any statement using an appropriate opcode.

In general, the location reserved for a variable in any assembly-language program may wind up anywhere in memory. This means that the address of the location may require two bytes to store. Direct memory addressing always uses two full bytes in the instruction to store the address:

opcode low-address byte high-address byte

The addresses of all locations in Page Zero have the property that they may be stored in a single byte. Put another way, the high-address byte for such a location is always zero. The Page Zero addressing mode takes advantage of this by not storing the high-address byte (or zero) in the instruction. Instructions which use Page Zero addressing only require two bytes, instead of three. This represents a 33 percent savings in space. No wonder all those APPLE systems programs—Monitor, DOS, APPLESOFT, etc.—gobble up Page Zero space like it was going out of style.

Use a section of Page Zero equates

A convenient way to declare variables that you intend to be on Page Zero is with the pseudo-op EQU. Every assembly code program should begin with a section of these pseudo-ops. In addition to declaring Page Zero variables, these equates may create symbolic identifiers for other constant values, external addresses, etc. The section of equates is analogous to the declaration section recommended in Chapter 2 for Integer BASIC programs.

Indexed Addressing

The indexed addressing mode may be used to access any of a sequence of bytes up to 255 bytes away from a given memory location. The technique is similar to indexing an array in Pascal or BASIC. The major differences are:

- Assembly language arrays are always 0-indexed (See Chapter 3).
- 6502 indexing is limited to a range of 255 bytes.
- 6502 indexes are always considered to be *positive* values.

The syntax for indexed addressing is as follows, using LDA as a sample instruction:

LDA ARRAY,X or **LDA ARRAY,Y**

depending on whether the X or Y register is used. Both direct and Page Zero indexed modes are available in the 6502 with the same distinction (two-byte instructions versus three-byte instructions) as for the nonindexed addressing modes. Most assemblers will choose the Page Zero mode if appropriate without explicit directions from the programmer.

Indirect-Indexed Addressing

This mode is similar to the plain indexed addressing just described. The difference is that instead of the instruction specifying the address of the array directly, it specifies a pair of Page Zero memory locations which contain the array address. That is, the instruction using indirect-

indexed addressing specifies a Page Zero address. At that address and the following address are two bytes which form a full 16-bit address. *That* address tells us the location relative to which the indexing is to take place. This mode requires the use of the Y register. There is another indirect mode which is less useful that employs the X register.

The syntax is as follows:

LDA (PTR),Y

where (PTR, PTR+1) is the pair of Page Zero locations containing the (indirect) pointer to the array to be indexed.

The advantages of the indirect-indexed mode over the plain indexed mode are many. Among them are:

- Instructions using it take only two bytes.
- It allows the same section of code to access, at least potentially, many different arrays.
- It allows access to more than 255 bytes of memory, by altering the Page Zero address, rather than the contents of the Y register.

Later in this chapter, we present an application which uses both plain and indirect-indexed addressing.

4. THE APPLE II OUTPUT HOOK AND COUT ROUTINE

The APPLE II Monitor and Autostart ROMs both contain a routine commonly referred to as COUT. COUT stands for Character OUT and is the standard routine to call when sending a character to the screen or other output device.

The Output Hook—CSWL,CSWH

The COUT routine begins with the interesting instruction:

JMP (CSWL)

This instruction uses the simple indirect addressing mode. This mode is only applicable to the JMP instruction and causes the program to transfer control to whatever address is stored in the Page Zero locations CSWL,CSWH (=$36,$37). This pair of locations is called the *output hook* and allows the actual output routine in effect at any given time to be varied.

On an ordinary APPLE II or APPLE II+ without any disk drives attached, the contents of the output hook will simply be the address of the instruction following JMP (CSWL). In the Monitor listings, this location is referred to as COUT1. The code at and following COUT1 processes the output character and calls the routine VIDOUT, which sends the character to the APPLE screen. Therefore, until some other program changes the contents of (CSWL,CSWH) any call to COUT merely causes a character to be sent to the APPLE screen.

Using the Hook

Placing a different address in (CSWL,CSWH) is known as "hooking" up a new output routine. That routine then gets control whenever COUT is called. Since the instruction which gives it (the new hook routine) control is a JMP instruction, the new routine may avoid going through COUT1 by simply performing a RTS instruction when it finishes. This is what happens when you type ↑I 8ON after a PR#1 turns on a printer. In more detail:

- PR#1 causes the address of the firmware routine for slot 1 to be placed into the output hook. Assuming that a printer interface card is installed in slot 1, then the printer has been turned on by this action.
- ↑I 8ON causes the printer firmware to turn off the screen. That is, the firmware will not do a JMP COUT1 after sending a character to the printer.

The address of a user routine may be placed into the output hook as well. This may be accomplished in a couple of ways.

- It may be POKEd into $36 and $37 (decimal 54,55) from a running program. (Note: Don't try it from the keyboard; see Explorations.)
- It may be stored by an assembly code routine.

The use of the hook by user output routines is complicated by the existence of DOS.

DOS and the Output Hook

The APPLE DOS makes heavy use of the output hook. Since DOS came along *after* both Integer BASIC and APPLESOFT BASIC were created, there were no built-in statements in either language which allowed for the use of files. Since many APPLES were already "in the field" by that time and since one or both BASIC interpreters were supported by ROM-based programs, it was desirable not to have to change the interpreters themselves. Yet, it was still necessary to find a way for BASIC programs to use the DOS routines.

The solution to the problem was to have DOS patch itself into the output hook. Then it could "listen" for a special signal. It was sort of like putting a wiretap on a phone. DOS, of course, looked for a Control-D character

in order to recognize commands to itself. Other output characters needed to be let through and on to their ultimate destination. Thus, DOS worked out a complex scheme of hooking and unhooking itself to allow maximum flexibility. We shall not try to unravel all that complexity here. However, just knowing that DOS is doing all this behind the scenes can make certain actions of sample programs more understandable.

Hooking and Unhooking

The following brief sections of code give the prototype methods for hooking and unhooking a routine. The label HOOK stands for a generic user output routine that is being inserted into the output hook.

- Hooking:

```
LDA >HOOK
STA CSWL
LDA <HOOK
STA CSWH
JSR $3EA
```

- Unhooking

```
JSR $FE93
JSR $3EA
```

Double Routine

Listing 18.1 presents a short assembly code routine which may be used to demonstrate the use of the COUT hook. This routine causes each character typed at the keyboard or otherwise sent to the screen to be printed twice. This routine serves mainly an educational function, although it can be fun to spring it on the unsuspecting. It may be invoked by the HELLO program on a given disk providing surprise and frustration for the unsuspecting.

Musical Keyboard

Listing 18.2 presents another humorous application of the COUT hook. This routine uses the numeric value of each key entered as a parameter for a call to the tone-playing routine in the Programmer's Aid (PA) ROM. Note: This comes for free in the RAM-based version of Integer BASIC which is loaded on systems with the language card. Otherwise, you will have to own the PA ROM to use this demo.

5. SENDING MESSAGES TO THE APPLE II SCREEN

Almost any program we write, be it in BASIC or Pascal or ancient Babylonian, will wish to interact with the user via messages displayed on the screen. Since assembly language does not have a PRINT or Writeln or whatever the ancient Babylonian equivalent would be, we must synthesize this capability out of simpler ingredients. Points which must be considered are:

- How do we represent and generate strings in assembly language?
- How do we PRINT a single string on the screen?
- What is the best way to keep track of several *different* messages?

Representation of Strings

To simplify our discussion, we consider each message to be a complete line of text which will terminate with a carriage return. With that in mind, each message may be declared using the ASC pseudo-op:

```
MSG1    ASC /*********/
          0
```

In order for our assembly code PRINT routine to be able to recognize the end of the string, we follow it by a 0 in memory.

All messages may be gathered together into a single part of the program and declared in a similar fashion:

```
MSG1    ASC /********************/
        0
MSG2    ASC /BUTTON,BUTTON/
        0
MSG3    ASC /WHO'S GOT THE BUTTON?/
        0
MSG4    ASC /THAT'S ALL FOLKS!/
        0
```

Printing a Single String

Each string may be sent to the screen a character at a time by calling COUT. The access to each string will be via indirect-indexed addressing. A pair of Page Zero locations may be reserved to hold a pointer to the string. These locations are set up with the address of the string prior to a call to the routine which prints the string. The following code may then be used to print the string:

```
OUTTXT  TYA             ;SAVE Y-REG
        PHA             ;ON 6502 STACK
        LDA $00         ;USE 0
                        AS INDEX
OLOOP   LDA (MSGPTR),Y  ;FOR INDIRECT
                        ADDRESSING
        BEQ OQUIT       ;TEXT ENDS
                        WITH A NUL
        JSR COUT        ;OUTPUT THE
                        CHARACTER
        INC MSGPTR      ;STEP POINTER
                        ALONG
        BNE OLOOP       ;STRING
        INC MSGPTR+1
        JMP OLOOP
OQUIT   PLA             ;GET OLD
                        Y-REG VALUE
        TAY             ;FROM STACK
                        AND PUT IT
                        IN Y
        RTS             ;RETURN FROM
                        SUBROUTINE
```

Saving registers in subroutines

In the above example, we used the Y register. Since the code above is intended to form a subroutine, there is no guarantee that the caller of the subroutine is not using the Y register for some other purpose. Therefore, the wise thing to do is to *save* the value of the Y register upon entrance to the subroutine. This is accomplished by putting the Y register into the accumulator and pushing it onto the stack. At the exit from the subroutine, the process is reversed: the value is "pulled" from the stack and then transferred back into the Y register.

The above process assumes that the accumulator is expendable. If that were not the case, then it would be necessary to save the accumulator *first:*

```
PHA         ;SAVE ACCUMULATOR
TYA         ;SAVE Y-REGISTER ON
PHA         ;6502 STACK
...         ;
...         ;SUBROUTINE
...         ;
PLA         ;GET Y-REGISTER VALUE
TAY         ;BACK FROM STACK
PLA         ;RETRIEVE ACCUMULATOR VALUE
```

Setting up MSGPTR

The above subroutine uses the Page Zero pointer (MSGPTR,MSGPTR+1). The caller of the string print subroutine would be responsible for setting up this pointer. In a program that uses multiple messages, it would be reasonable to store the addresses of the messages in a table. The caller of the message print routine could first pluck the appropriate address from this table.

In order to retrieve the *n*th entry in a table of addresses, it is reasonable to use indexed addressing. Since the contents of the table consists of 16-bit or two-byte quantities, it is necessary first to double the index into the table:

```
LDA MSGNUM      ;PREPARE TO GET
                ADDRESS OF
ROL A           ;MSGNUM'TH MESSAGE —
                DOUBLE
TAX             ;THE INDEX AND PUT
                IN X-REG
JSR GETPTR      ;CALL POINTER ACCESS
                ROUTINE
```

The value in the X register would then be two times the message number. Assuming that the messages are numbered beginning at 0, this would give the location within the table of addresses of the first byte of the msgnum'th message address. The code for copying that address into the Page Zero pointer would then be:

```
GETPTR  LDA MSGTAB,X    ;GET LOW BYTE
                        OF ADDRESS
        STA MSGPTR      ;PUT IN LOW BYTE
                        OF PZ PTR
        INX             ;POINT TO HIGH BYTE
        LDA MSGTAB,X    ;GET HIGH BYTE
                        OF ADDRESS
        STA MSGPTR+1    ;PUT IN HIGH BYTE
                        OF PZ PTR
        RTS             ;GO BACK TO CALLER
```

The message table may be formed using the DW pseudo-op:

```
MSGTAB  DW      MSG1
        DW      MSG2
        DW      MSG3
        ...
        ETC.
```

In Chapter 19, we present a program that applies these techniques to produce a series of "help" messages on the screen.

6. EXPLORATIONS

- Expand the cryptographic hook routine to use a simple substitution cipher. This means that each character in the coded text will be a substitute for the corresponding character in the plain text. The assembly code to perform the substitution could use indexing to look up the code character in a table.
- What other applications can you devise for the output hook?
- Investigate the input hook on the APPLE. Write a pair of hook routines to encrypt and decrypt text files using techniques similar to those already explored.
- Read the code for the APPLE Monitor ROMs (in the APPLE II Reference Manual). What assembly language techniques can you recommend from this source?
- What will happen if you try to hook up a new output routine by using POKEs directly from the keyboard? Can you explain the reason for the results you describe?

LISTINGS

LISTING 18.1 DOUBLE TROUBLE KEYBOARD HOOK

```
FDF0:              2 COUT1    EQU  $FDF0      ;MONITOR CHARACTER OUTPUT ROUTINE
0036:              3 CSWL     EQU  $36
0037:              4 CSWH     EQU  $37
008C:              5 CTRLL    EQU  $8C
008D:              6 CTRLM    EQU  $8D
02FD:              7 VOICE    EQU  $2FD
02FE:              8 LONG     EQU  $2FE
02FF:              9 NOTE     EQU  $2FF
03EA:             10 MVSW     EQU  $3EA       ;ROUTINE WHICH RECONNECTS DOS
FE93:             11 SETVID   EQU  $FE93
----- NEXT OBJECT FILE NAME IS LISTING 18.1 *****.OBJ0
0300:             12          ORG  $300
0300:4C 0F 03     13          JMP  UNHOOK
0303:             14 ;***
0303:             15 ;***
                  16 ************************************************************
0303:             17 ;***
0303:             18 ;*** ROUTINE TO CONNECT US INTO OUTPUT HOOK
0303:             19 ;***
                  20 ************************************************************
0303:             21 ;***
0303:             22 ;***


0303:A9 16        24          LDA  #>DOUBLE   ;LOW BYTE OF ADDRESS
0305:85 36        25          STA  CSWL
0307:A9 03        26          LDA  #<DOUBLE    ;HIGH BYTE OF ADDRESS
0309:85 37        27          STA  CSWH
030B:20 EA 03     28          JSR  MVSW       ;TELL DOS OF OUR INTENTIONS
030E:60           29          RTS
030F:             30 ;***
030F:             31 ;***
                  32 ************************************************************
030F:             33 ;***
030F:             34 ;*** THIS UNHOOKS OUR ROUTINE
030F:             35 ;***
                  36 ************************************************************
030F:             37 ;***
030F:             38 ;***


030F:20 93 FE     40 UNHOOK   JSR  SETVID     ;SIMULATE PR#0
0312:20 EA 03     41          JSR  MVSW       ;RECONNECT DOS
0315:60           42          RTS
0316:             45 ;***
0316:             46 ;***
                  47 ************************************************************
0316:             48 ;***
0316:             49 ;***    D O U B L E   T R O U B L E
0316:             50 ;***
0316:             51 ;***  THIS ROUTINE CAUSES EACH CHARACTER SENT
0316:             52 ;*** TO THE SCREEN TO APPEAR TWICE.
0316:             53 ;*** IT DOES THIS BY INVOKING COUT1 TWICE
0316:             54 ;***
```

```
                        55 ************************************************************
0316:                   56 ;***
0316:                   57 ;***


0316:20 F0 FD           59 DOUBLE  JSR  COUT1
0319:4C F0 FD           60         JMP  COUT1

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

SYMBOL TABLE SORTED BY SYMBOL

```
 FDF0 COUT1              37 CSWH              36 CSWL              ?  8C CTRLL
?  8D CTRLM            0316 DOUBLE          ?02FE LONG             03EA MVSW
?02FF NOTE             FE93 SETVID            030F UNHOOK          ?02FD VOICE
```

SYMBOL TABLE SORTED BY ADDRESS

```
   36 CSWL              37 CSWH              ?  8C CTRLL          ?  8D CTRLM
?02FD VOICE            ?02FE LONG           ?02FF NOTE             030F UNHOOK
 0316 DOUBLE            03EA MVSW             FDF0 COUT1            FE93 SETVID
```

LISTING 18.2 MUSICAL KEYBOARD HOOK ROUTINE

```
FDF0:                    2 COUT1   EQU  $FDF0     ;MONITOR CHARACTER OUTPUT ROUTINE
0036:                    3 CSWL    EQU  $36
0037:                    4 CSWH    EQU  $37
008C:                    5 CTRLL   EQU  $8C
008D:                    6 CTRLM   EQU  $8D
02FD:                    7 VOICE   EQU  $2FD
02FE:                    8 LONG    EQU  $2FE
02FF:                    9 NOTE    EQU  $2FF
03EA:                   10 MVSW    EQU  $3EA      ;ROUTINE WHICH RECONNECTS DOS
FE93:                   11 SETVID  EQU  $FE93
D717:                   12 MUSIC   EQU  $D717
----- NEXT OBJECT FILE NAME IS LISTING 18.2 *****.OBJ0
0300:                   13         ORG  $300
0300:4C 19 03           14         JMP  UNHOOK
0303:                   15 ;***
0303:                   16 ;***
                        17 ************************************************************
0303:                   18 ;***
0303:                   19 ;*** ROUTINE TO CONNECT US INTO OUTPUT HOOK
0303:                   20 ;***
                        21 ************************************************************
0303:                   22 ;***
0303:                   23 ;***


0303:A9 20              25         LDA  #>PLAY    ;LOW BYTE OF ADDRESS
0305:85 36              26         STA  CSWL
0307:A9 03              27         LDA  #<PLAY    ;HIGH BYTE OF ADDRESS
0309:85 37              28         STA  CSWH
```

LISTING 18.2 (*cont.*)

```
030B:A9 40      29          LDA  #$40
030D:8D FD 02   30          STA  VOICE
0310:A9 0A      31          LDA  #$0A
0312:8D FE 02   32          STA  LONG
0315:20 EA 03   33          JSR  MVSW       ;TELL DOS OF OUR INTENTIONS
0318:60         34          RTS
0319:           35 ;***
0319:           36 ;***
                37 ************************************************************
0319:           38 ;***
0319:           39 ;*** THIS UNHOOKS OUR ROUTINE
0319:           40 ;***
                41 ************************************************************
0319:           42 ;***
0319:           43 ;***


0319:20 93 FE   45 UNHOOK   JSR  SETVID     ;SIMULATE PR#0
031C:20 EA 03   46          JSR  MVSW       ;RECONNECT DOS
031F:60         47          RTS
0320:           50 ;***
0320:           51 ;***
                52 ************************************************************
0320:           53 ;***
0320:           54 ;***      M U S I C A L   K E Y B O A R D
0320:           55 ;***
0320:           56 ;*** THIS ROUTINE USES THE KEY VALUE TO
0320:           57 ;*** DETERMINE A NOTE TO BE PLAYED.
0320:           58 ;*** IT CALLS THE MUSIC SUBROUTINE IN THE
0320:           59 ;*** PA ROM TO DO THIS.
0320:           60 ;***
                61 ************************************************************
0320:           62 ;***
0320:           63 ;***


0320:48         65 PLAY     PHA             ;SAVE PRINTABLE CHARACTER
0321:C9 32      66 PLAY1    CMP  #$32       ;IS IT TOO BIG?
0323:90 06      67          BCC  SOUND
0325:38         68          SEC             ;YES - CONVERT
0326:E9 50      69          SBC  #$50       ;TO LEGAL NOTE
0328:4C 21 03   70          JMP  PLAY1
032B:8D FF 02   71 SOUND    STA  NOTE
032E:20 17 D7   72          JSR  MUSIC
0331:68         73          PLA                 ;RESTORE CHARACTER
0332:4C F0 FD   74          JMP  COUT1

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

SYMBOL TABLE SORTED BY SYMBOL

```
 FDF0 COUT1          37 CSWH          36 CSWL          ?  8C CTRLL
?  8D CTRLM       02FE LONG        D717 MUSIC          03EA MVSW
 02FF NOTE        0320 PLAY        0321 PLAY1          FE93 SETVID
 032B SOUND       0319 UNHOOK      02FD VOICE
```

SYMBOL TABLE SORTED BY ADDRESS

```
  36 CSWL          37 CSWH          ?  8C CTRLL      ?  8D CTRLM
02FD VOICE       02FE LONG          02FF NOTE        0319 UNHOOK
0320 PLAY        0321 PLAY1         032B SOUND       03EA MVSW
D717 MUSIC       FDF0 COUT1         FE93 SETVID
```

Chapter

19

Ampersand Support Routines

In Chapter 11, we presented a version of the Giant Letters Screen program which used a machine-language program to implement the format strings interpreter of Chapter 3. In this chapter we study that program in detail. In addition, we present another ampersand support routine. It allows you to produce paginated listings of APPLESOFT programs. It exploits the APPLE II output hook introduced in the previous chapter.

In the process of detailing the ampersand support routines, we shall dwell on some internal aspects of APPLESOFT. In particular, we will learn how to manipulate APPLESOFT strings from an assembly-language program. We will also learn how to extract characters and/or other information from the APPLESOFT program.

1. THE AMPERSAND COMMAND REVISITED

We learned earlier that when APPLESOFT executes the & command it makes an unconditional jump to memory location $3F5. This means that whatever instruction is located at $3F5 will be executed next. Since we can put whatever instruction we wish into $3F5, this means that we have wrested control from the APPLESOFT program.

What to put into $F35

There is really only one practical answer to this question: A jump instruction that "vectors" to our own machine-language routine. Let's explore why.

"Vectors" in the APPLE

Many people toss around the term "vector." Few bother to define it. Let's join the few. A machine language *vector* is simply a jump instruction. It transfers control, or *vectors* the CPU to another place in memory. This terminology isn't limited to the machine-language programming world. We also find it in the world of air traffic control. Controllers give airliners vectors to an approach heading. In plain English, this means that the airplane is being pointed in a series of different directions. Each direction is one vector.

Likewise, a machine-language vector simply routes the CPU to a different place in memory for the next instruction. It tells the CPU where to go.

A natural reaction at this point is again to ask *why?* It would seem that vectors simply waste time. They introduce an extra jump instruction where none is evidently needed.

Strictly speaking, vectors are not ever needed. But they turn out to increase the flexibility of software that uses them. Let's look at some vectors that are used by APPLE's own DOS and see if we can gain some insight.

DOS Vectors

The reason that we can't simply put a complete machine-language routine beginning at $3F5 is that that location is right smack in the middle of a *series* of vectors. APPLE DOS 3.2 and 3.3 both store various vectors in memory locations $3D0 to $3FF. One of these is the ampersand command vector at $3F5.

In order to allow ampersand support routines to exist *anywhere* in memory, the vector concept is used. In location $3F5 we can put:

```
JMP $9000
JMP $800
JMP $513F
```

or whatever we please. We only have to *permanently dedicate* three bytes in order to support ampersand. Suppose APPLESOFT had decided to try to jump *directly* to the ampersand support routine. Then it would have been necessary to choose a location where we could guarantee enough free bytes of memory to allow a "substantial" support routine. But what do we mean by "substantial"? 1000 bytes? 3000 bytes? 256 bytes? No two programmers would give the same answer to that one. No, it is better to let the programmer choose where to locate the routine. Then it's up to him or her to provide the space.

In addition to the ampersand vector, DOS puts jumps to other "entry points" in itself in the locations on page 3. For example, at $3D0 is a jump to the beginning of the code for DOS itself. This is the so-called warm-start entry point. Most APPLE II programmers are familiar with the command:

```
3D0G
```

typed to the monitor to restart the DOS after accidental or intentional interruptions.

Since the warm-start entry point to DOS depends on the amount of memory available on the particular APPLE, the vector is provided to make the command 3D0G independent of memory configuration. Putting it another way: in order to allow the DOS the flexibility to exist at different addresses on differently sized APPLEs, it is necessary to use the vector. Figure 19.1 illustrates the situation.

2. SOME USEFUL APPLESOFT ROUTINES

In order to write interesting ampersand (&) support routines, it is necessary to delve into the inner workings of APPLESOFT. We first examine the way the program is represented in APPLE RAM.

APPLESOFT Program Representation

The APPLESOFT interpreter translates each source program statement into a *tokenized* internal form. Keywords such as IF, INPUT, GOTO, PRINT, etc., are assigned single byte numeric values called tokens (see Chapters 14 and 15). Some single characters are represented by their ASCII values and others such as operators, etc., are represented by APPLESOFT tokens.

Consider the following statements:

```
100 POKE 253,0:POKE 254,0
105 GOSUB 1000
110 COLOR=7:     B$="H050V050"
115 &L&(C)
```

They would be translated by APPLESOFT into the internal forms shown in Figure 19.2. There are several points to be made about this example:

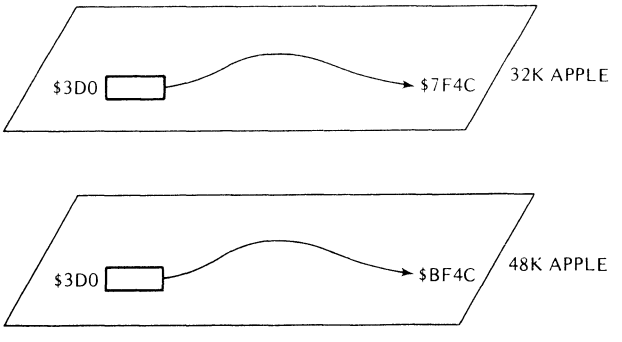
- Keywords have been tokenized:

  POKE -> B9
  GOSUB -> B0
  COLOR = -> A0
  & -> AF

- The names of variables are represented by ASCII character values:

  B$ -> 42 24
  L$(C) -> 4C 24 28 43 29

FIGURE 19.1 Interpretation of $3D0 Vector on APPLE's with Different Sizes of Memory



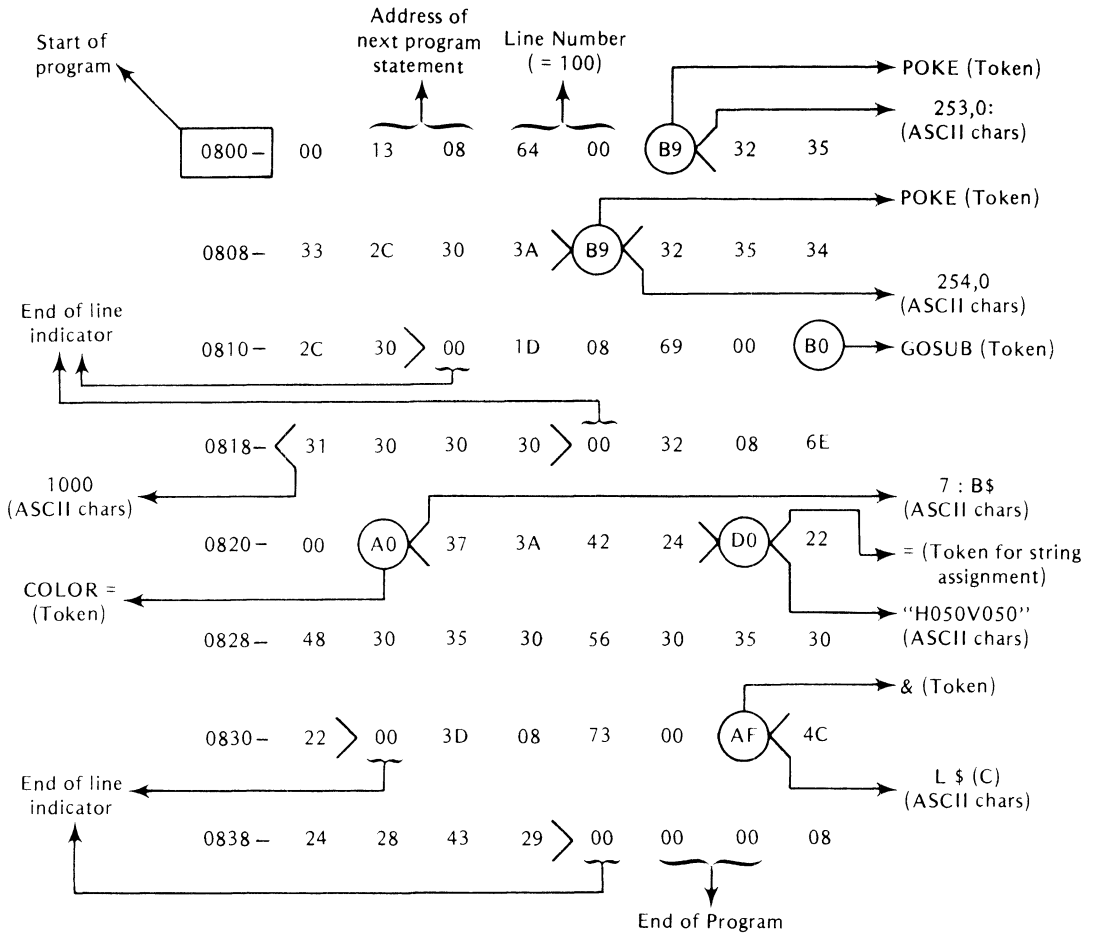


FIGURE 19.2 APPLESOFT Program Representation

- Blanks which are not part of a string are removed by the APPLESOFT interpreter.
- Character strings are represented literally by the corresponding sequence of ASCII values:

  "H050V050" → 22 48 30 35 30 56 30 35 30 22

- The end of each statement is marked by a 00.

Retrieving Characters from APPLESOFT Programs

APPLESOFT uses one routine with great frequency. In fact, it is used so often that it is located on Page Zero. The code is as follows:

```
00B1: INC $B8      ;BUMP CHAR ADDR (LOW)
00B3: BNE $00B7    ;CHECK FOR CARRY
00B5: INC $B9      ;BUMP CHAR ADDR (HIGH)
00B7: LDA $0205    ;GET CHARACTER
00BA: CMP #$3A     ;IS IT <= ":"?
00BC: BCS $00C8    ;YES — GO BACK
00BE: CMP #$20     ;NO — CHECK FOR BLANK
00C0: BEQ $00B1    ;SKIP IF FOUND
00C2: SEC          ;ADJUST FOR INTERNAL
00C3: SBC #$30     ;REPRESENTATION
00C5: SEC
00C6: SBC #$D0
00C8: RTS          ;GO BACK
```

This routine is referred to in the APPLE literature as CHRGET. Its function is to return the next character or token from the "current source." The current source may be the APPLESOFT program or the keyboard input buffer (located on page 2 starting at $200). APPLESOFT patches the CHRGET routine when the source of characters switches from one place to another. For example, when the user types a RUN command, the source of characters switches from the keyboard input buffer to the first character of the APPLESOFT source program. The new address is plunked into locations $B8 and $B9.

The portion of the CHRGET routine beginning at $B7 is also callable and is known as CHRGOT. It is used whenever it is necessary to reexamine the most recently retrieved character. It is called if that character which was in the AC gets lost or destroyed.

The application of CHRGET in ampersand routines is to pick characters from the BASIC program. Each call to CHRGET advances to the next character, allowing the user to build any scanning routines desired. One point to note is that when the & command is executed, APPLESOFT itself calls CHRGET *before* vectoring through $3F5. Therefore, when the machine language program is entered, the next character is *already* in the accumulator. The CHRGOT routine is frequently used at the end of an ampersand routine. It is called at the beginning of a series of statements designed to scan to the end of the source program line. Since the 00 which marks the end of the line may already have been retrieved by the main program and rejected as not a legal command, CHRGOT is called instead of CHRGET.

Retrieving Strings from APPLESOFT Programs

In our first application of the ampersand command we will need to locate the *value* of a string variable. However, only the *name* of the variable is available in the source:

&B$** or **&L$(C)

Fortunately APPLESOFT again provides the answer. This time it provides us with the routine PTRGET.

PTRGET is called when the CHRGET character address is positioned at the name of an APPLESOFT variable:

&B$** or **&L$(C)

The routine returns the address of the value of the variable. This address is returned by PTRGET as the contents of the A and Y registers. The low byte of the address is returned in A. The high byte of the address is returned in Y.

The address returned by PTRGET for a floating-point variable is the address of the location at which the value of the variable is stored. The address returned for a string variable is that of a *descriptor* of the string's value. A *string descriptor* is a three-byte sequence containing two pieces of information:

BYTE 1: The length of the string in characters.
BYTE 2: } The address of the characters of the
BYTE 3: } string.

Figure 19.3 illustrates string descriptors and the action of PTRGET.

A string which has been assigned as the value of a string variable is terminated by a binary 00. On the other hand, strings which are embedded as constants, for example, as the argument of a PRINT statement, are terminated by either a double quote (") or a 00.

10 PRINT "HI" → ends with "
10 PRINT "HI → ends with 00

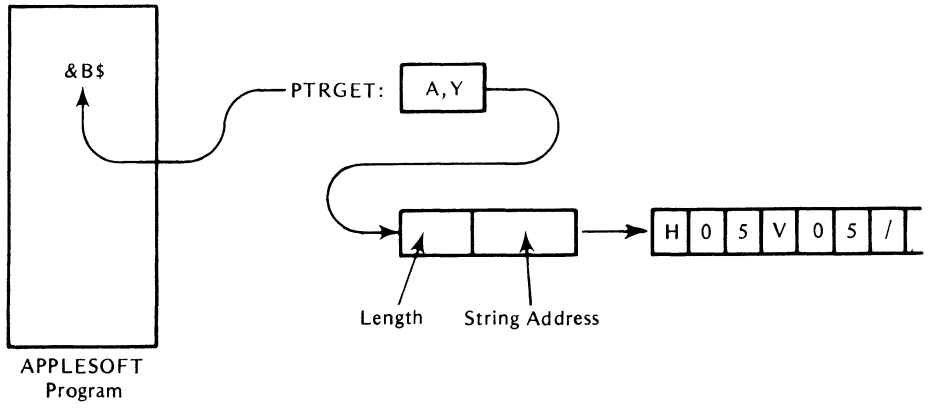
3. AMPER-GRAPHICS

Listing 19.1 presents a format string interpreter. It is written in assembly language and is intended to be invoked by an & command. The program assumes that the *name* of a string variable follows the &. If the user writes a constant string:

&"H050V050"

the Amper-Graphics routines will flag a SYNTAX ERROR.

FIGURE 19.3 Use of String Descriptors and PTRGET



Retrieving the String Descriptor

Amper-Graphics' first act is to call PTRGET. The returned address is stored in the Page Zero pointer (DESC,DESC+1). The pointer is used via indirect-indexed addressing to retrieve the length and address of the string value.

The address of the characters of the string is stored in the Page Zero pointer (CHARADDR,CHARADDR+1). This pointer is used in the getchar routine to return the individual characters of the format string.

Getchar

Since the format strings passed to Amper-Graphics are not in general embedded in the source program, CHRGET will not suffice to retrieve them. The getchar routine was written to remedy this. Getchar uses the (CHARADDR,CHARADDR+1) pointer to access the characters of the string. Most of the other parts of the Amper-Graphics program rely on getchar.

The Main-Line Code

The main code of the Amper-Graphics routine acts like a Pascal case statement. The case discrimination is based on the character from the format string. Compare this code with the code in the Integer BASIC version of the program in Chapter 3.

Support Routines

Each case in the main-line code invokes a separate Amper-Graphics support routine. These routines serve to implement the individual graphics commands such as PLOT, HLINE, TRANSLATE, FILL, etc. Each of these routines first makes calls to getchar to retrieve its arguments. It then carries out its individual task.

4. AMPER-LIST

The program of Listing 19.2 implements another & support routine. It may be used to list APPLESOFT programs. It is an extension of a simpler program. The simpler program was developed to help a printer skip over the perforations separating individual pages of a listing. The printer in question lacked the capacity to do that by itself. The fundamental capability of Amper-List is therefore to count lines on a page and leave a margin at the top and bottom of each.

Amper-List Capabilities and Design

Amper-List uses the Monitor output hook at $36 and $37. It *assumes* that it is running under DOS. Each command to Amper-List takes the form of a single character following the &. These are recognized by Amper-List by the use of simple CMP instructions. APPLESOFT calls CHRGET for us and therefore we may assume that the command letter is already in the AC upon entry to Amper-List. The command letters are as follows:

I	Initialize
U	Unhook
F	Form feed
%"	Title

The meaning and use of each is now detailed:

&I This turns on the printer by calling the initialization code in the printer card firmware. We have coded Amper-List to assume that the printer card is in slot 1. In addition, various internal quantities are set up. Namely, the line count per page is set to 0; the page number is set to 1, and the title buffer is blanked. Finally, the output hook is set to the address of the main Amper-List code and the DOS is so notified.

&U This turns off the printer and unhooks the print over perforations hook routine.

&F This issues a form-feed to the printer. It also resets the page number in the title buffer to 1. This allows the user of the routine to generate a fresh page on the printer just before issuing the LIST command.

&" This indicates that a title is to follow. The characters that follow are scanned until another " is encountered. They are copied into the title buffer and will subsequently be used to print a title at the top of each page of the listing. Notice that the program does not protect itself against the silly user who forgets to end the title string with a ". This oversight is easy to remedy and we have left that to you (see Explorations).

Typical Use of Amper-List

To use Amper-List, it is a good idea to set up an EXEC file. It should contain at least the following:

```
BLOAD AMPER-LIST.OBJ0
BLOAD AMPER-GRAPHICS.OBJ1
```

The second BLOAD causes the vector instruction: JMP $9000 to be loaded into $3F5–$3F7. Once the two binary files above have been loaded, then a typical sequence might be as follows:

```
&I
&"GIANT LETTERS:  APPLESOFT VERSION"
^I80N
&F
LIST
```

This sequence turns on the program, sets up "GIANT LETTERS: APPLESOFT Version" as a title, asks the interface card to print 80 characters per line, asks the program for a fresh page, and finally lists the program. This exact sequence was used to generate Listing 11.1. (Also see the Explorations section.)

Using Amper-List in a Program

Amper-List may also be used in a running APPLESOFT program. Listing 19.3 gives an example of such an application. It was used to generate Listing 11.3.

5. EXPLORATIONS

- Add error handling code to Amper-List to catch any title commands that fail to end in a ".
- Investigate APPLESOFT tokens more carefully. See if you can explain why CHRGET routines do the two subtractions.
- Devise a way to have more than two ampersand routines in memory at once. What is a good way to select between them? Can you think of a way to have more than the ampersand vector at $3F5?
- Design and implement other Ampersand utilities.
- Modify Amper-List so that the F command causes the title TO be listed on the first page.
- What happens when you write:

  CALL 1013". . ."

  in an APPLESOFT program (1013 decimal = $3F5). Can you think of ways to make use of this behavior?
- Use indexed addressing to rewrite Amper-List so that it does not depend on the particular slot that the printer happens to be in.
- Add new commands to the Amper-Graphics routines.

LISTINGS

LISTING 19.1 MAIN DRIVER

```
0000:              2          MSB  OFF
0006:              3 DESC     EQU  $6
0008:              4 CHARADDR EQU  $8
00D0:              5 ERRFLG   EQU  $D0
00F9:              6 RX       EQU  $F9
00FA:              7 CX       EQU  $FA
00FB:              8 AX       EQU  $FB
00FC:              9 BX       EQU  $FC
00FD:             10 ROW      EQU  $FD
00FE:             11 COL      EQU  $FE
00FF:             12 CHARS    EQU  $FF
002C:             13 H2       EQU  $2C
002D:             14 V2       EQU  $2D
00B1:             15 CHRGET   EQU  $B1
00B7:             16 CHRGOT   EQU  $B7
D412:             17 ERROR    EQU  $D412
DFE3:             18 PTRGET   EQU  $DFE3
F800:             19 PLOT     EQU  $F800
F819:             20 HLINE    EQU  $F819
F828:             21 VLINE    EQU  $F828
F864:             22 SETCOL   EQU  $F864

0000:             25 ;***
0000:             26 ;***
                  27 ************************************************************
0000:             28 ;***
0000:             29 ;***   A  M  P  E  R  -  G  R  A  P  H  I  C  S
0000:             30 ;***
0000:             31 ;***  THESE ROUTINES MAY BE CALLED FROM AN
0000:             32 ;***  APPLESOFT PROGRAM TO CREATE LOW-RES
0000:             33 ;***  GRAPHICS DISPLAYS.
0000:             34 ;***
                  35 ************************************************************
0000:             36 ;***
0000:             37 ;***


------ NEXT OBJECT FILE NAME IS LISTING 19.1 *****.OBJO
9000:             39          ORG  $9000
9000:20 E3 DF     40 START    JSR  PTRGET     ;GET POINTER TO STRING VAR.
9003:85 06        41          STA  DESC       ;SAVE IN PAGE ZERO POINTER
9005:84 07        42          STY  DESC+1
9007:A0 00        43          LDY  #$00       ;USE DESC POINTER
9009:B1 06        44          LDA  (DESC),Y   ;INDEXED BY Y TO PICK UP
900B:85 FF        45          STA  CHARS      ;LENGTH OF STRING
900D:E6 FF        46          INC  CHARS      ;(STORE AS LEN + 1)
900F:C8           47          INY              ;AND
9010:B1 06        48          LDA  (DESC),Y   ;ADDRESS OF THE STRING
9012:85 08        49          STA  CHARADDR   ;OF CHARACTERS ITSELF
9014:C8           50          INY              ;SAVE THE LATTER
9015:B1 06        51          LDA  (DESC),Y   ;IN A PAGE ZERO
```

```
9017:85 09       52           STA  CHARADDR+1 ;POINTER AS WELL
9019:20 93 90    53 DECODE    JSR  GETCHAR
901C:F0 60       54           BEQ  DONE
901E:C9 48       55           CMP  #'H       ;CHECK FOR HLINE COMMAND
9020:D0 06       56           BNE  DEC1      ;NOT IT-TRY NEXT
9022:20 AD 90    57           JSR  HRZNTL    ;PROCESS THE "H" COMMAND
9025:4C 19 90    58           JMP  DECODE    ;CONTINUE INTERPRETATION
9028:C9 56       59 DEC1      CMP  #'V       ;CHECK FOR VLINE
902A:D0 06       60           BNE  DEC2      ;NOPE-TRY NEXT
902C:20 D0 90    61           JSR  VRTCL     ;PROCESS "V" COMMAND
902F:4C 19 90    62           JMP  DECODE    ;CONTINUE
9032:C9 50       63 DEC2      CMP  #'P       ;CHECK FOR PLOT
9034:D0 06       64           BNE  DEC3      ;NOT PLOT
9036:20 F3 90    65           JSR  PLT       ;PROCESS "P" COMMAND
9039:4C 19 90    66           JMP  DECODE    ;CONTINUE
903C:C9 43       67 DEC3      CMP  #'C       ;CHECK FOR COLOR=
903E:D0 06       68           BNE  DEC4      ;HO-HUM - NOT YET!
9040:20 0B 91    69           JSR  COLR      ;PROCESS "C" COMMAND
9043:4C 19 90    70           JMP  DECODE    ;CONTINUE
9046:C9 54       71 DEC4      CMP  #'T       ;CHECK FOR TRANSLATE
9048:D0 06       72           BNE  DEC5      ;BE PERSISTENT
904A:20 12 91    73           JSR  TRNSLTE   ;PROCESS "T" COMMAND
904D:4C 19 90    74           JMP  DECODE    ;CONTINUE
9050:C9 44       75 DEC5      CMP  #'D       ;CHECK FOR DIAGONAL
9052:D0 06       76           BNE  DEC6      ;IT'S LIKE PULLING TEETH
9054:20 34 91    77           JSR  DIAGON    ;PROCESS "D" COMMAND
9057:4C 19 90    78           JMP  DECODE    ;CONTINUE
905A:C9 42       79 DEC6      CMP  #'B       ;CHECK FOR BLOCK
905C:D0 06       80           BNE  DEC7      ;HOW MANY OF THESE ARE THERE?
905E:20 77 91    81           JSR  BLOCK     ;PROCESS "B" COMMAND
9061:4C 19 90    82           JMP  DECODE    ;CONTINUE
9064:C9 53       83 DEC7      CMP  #'S       ;CHECK FOR SLIDE
9066:D0 06       84           BNE  DEC8      ;CAN'T BE MANY MORE!
9068:20 23 91    85           JSR  SLIDE     ;PROCESS "S" COMMAND
906B:4C 19 90    86           JMP  DECODE    ;CONTINUE
906E:C9 46       87 DEC8      CMP  #'F       ;CHECK FOR FILL
9070:D0 06       88           BNE  ERR       ;WELL THAT'S IT - GIVE UP
9072:20 52 91    89           JSR  FILL      ;PROCESS "F" COMMAND
9075:4C 19 90    90           JMP  DECODE    ;CONTINUE
9078:A2 10       91 ERR       LDX  #$10      ;SIGNAL SYNTAX ERROR
907A:A9 FF       92           LDA  #$FF
907C:85 D0       93           STA  ERRFLG    ;TELL APPLESOFT
907E:20 B7 00    94 DONE      JSR  CHRGOT    ;CHECK OUT PREVIOUS CHARACTER
9081:C9 00       95 DONE1     CMP  #$00      ;IS IT THE END?
9083:F0 06       96           BEQ  AQUIT     ;YES- GIVE UP
9085:20 B1 00    97           JSR  CHRGET    ;NO - LOOK SOME MORE
9088:4C 81 90    98           JMP  DONE1     ;TRY AGAIN
908B:A5 D0       99 AQUIT     LDA  ERRFLG    ;WAS THERE AN ERROR?
908D:F0 03      100           BEQ  ARTS      ;NO - NORMAL RETURN
908F:4C 12 D4   101           JMP  ERROR       ;YES - TELL USER
9092:60         102 ARTS      RTS            ;AMPER-QUIT
```

LISTING 19.1 *(cont.)*

```
9093:            106 ;***
9093:            107 ;***
                 108 ************************************************************
9093:            109 ;***
9093:            110 ;***     G  E  T  C  H  A  R
9093:            111 ;***
9093:            112 ;*** RETRIEVE NEXT CHARACTER FROM AN AMPER
9093:            113 ;*** GRAPHICS CODED STRING.  USED INSTEAD
9093:            114 ;*** OF APPLESOFT CHRGET, SINCE THE THE
9093:            115 ;*** CODED STRINGS MAY NOT HAVE TRAILING
9093:            116 ;*** DOUBLE-QUOTES.
9093:            117 ;***
                 118 ************************************************************
9093:            119 ;***
9093:            120 ;***


9093:A0 00       122 GETCHAR LDY   #$00
9095:B1 08       123         LDA   (CHARADDR),Y
9097:E6 08       124         INC   CHARADDR
9099:D0 02       125         BNE   GCRTS
909B:E6 09       126         INC   CHARADDR+1
909D:C6 FF       127 GCRTS   DEC   CHARS
909F:60          128         RTS
```

LISTING 19.1a GETCHAR SUBROUTINE

```
90A0:            132 ;***
90A0:            133 ;***
                 134 ************************************************************
90A0:            135 ;***
90A0:            136 ;***     G  E  T  A  R  G
90A0:            137 ;***
90A0:            138 ;***  ROUTINE WHICH PLUCKS ARGUMENTS FROM
90A0:            139 ;***  THE APPLESOFT PROGRAM LINE.
90A0:            140 ;***
                 141 ************************************************************
90A0:            142 ;***
90A0:            143 ;***


90A0:20 93 90    145 GETARG  JSR   GETCHAR    ;GET ARGUMENT
90A3:38          146         SEC
90A4:E9 30       147         SBC   #$30       ;IS IT A DIGIT?
90A6:C9 09       148         CMP   #$09       ;CHECK
90A8:90 02       149         BCC   GA1        ;CARRY CLEAR IF <= 9
90AA:E9 07       150         SBC   #$07       ;CONVERT LETTER ARG. TO BINARY
90AC:60          151 GA1     RTS              ;ARGUMENT RETURNED IN AC
```

LISTING 19.1b "H" COMMAND HANDLER

```
90AD:            155 ;***
90AD:            156 ;***
                 157 ************************************************************
90AD:            158 ;***
90AD:            159 ;*** PROCESS THE "H" COMMAND: HXYZ
90AD:            160 ;***
90AD:            161 ;*** ARGUMENTS ARE - X = STARTING COLUMN OFFSET
90AD:            162 ;***                 Y = ENDING COLUMN OFFSET
90AD:            163 ;***                 Z = ROW OFFSET
90AD:            164 ;***
                 165 ************************************************************
90AD:            166 ;***
90AD:            167 ;***


90AD:20 A0 90    169 HRZNTL  JSR  GETARG     ;PICK UP FIRST ARGUMENT
90B0:85 F9       170         STA  RX         ;SAVE IT
90B2:20 A0 90    171         JSR  GETARG     ;GET 2ND ARG
90B5:85 FA       172         STA  CX         ;SAVE IT
90B7:20 A0 90    173         JSR  GETARG
90BA:85 FB       174         STA  AX
90BC:18          175 HRZIND  CLC             ;NORMALIZE
90BD:A5 FE       176         LDA  COL        ;GET CURRENT COL#
90BF:65 FA       177         ADC  CX         ;ADD IN ENDING OFFSET
90C1:85 2C       178         STA  H2         ;SAVE IN MONITOR LOC
90C3:A5 FE       179         LDA  COL        ;GET COL# BACK
90C5:65 F9       180         ADC  RX         ;ADD IN STARTING OFFSET
90C7:A8          181         TAY             ;SET UP ARG. TO PLOT
90C8:A5 FD       182         LDA  ROW        ;GET CURRENT ROW#
90CA:65 FB       183         ADC  AX         ;ADD OFFSET
90CC:20 19 F8    184         JSR  HLINE      ;DRAW THE LINE
90CF:60          185         RTS             ;BYE-BYE
```

LISTING 19.1c "V" COMMAND HANDLER

```
90D0:            189 ;***
90D0:            190 ;***
                 191 ************************************************************
90D0:            192 ;***
90D0:            193 ;*** PROCESS THE "V" COMMAND: VXYZ
90D0:            194 ;***
90D0:            195 ;*** ARGUMENTS ARE - X = STARTING ROW OFFSET
90D0:            196 ;***                 Y = ENDING ROW OFFSET
90D0:            197 ;***                 Z = COLUMN OFFSET
90D0:            198 ;***
                 199 ************************************************************
90D0:            200 ;***
90D0:            201 ;***
```

LISTING 19.1 *(cont.)*

```
90D0:20 A0 90   203 VRTCL   JSR  GETARG
90D3:85 F9      204         STA  RX
90D5:20 A0 90   205         JSR  GETARG
90D8:85 FA      206         STA  CX
90DA:20 A0 90   207         JSR  GETARG
90DD:85 FB      208         STA  AX
90DF:18         209         CLC              ;PREPARE ADDITION
90E0:A5 FD      210         LDA  ROW         ;GET ROW#
90E2:65 FA      211         ADC  CX          ;ADD ENDING OFFSET
90E4:85 2D      212         STA  V2          ;SAVE IN MONITOR LOC.
90E6:A5 FE      213         LDA  COL         ;GET CURRENT COL#
90E8:65 FB      214         ADC  AX          ;ADD IN OFFSET
90EA:A8         215         TAY              ;PREPARE CALL TO VLINE
90EB:A5 FD      216         LDA  ROW         ;GET ROW# BACK
90ED:65 F9      217         ADC  RX          ;ADD IN STARTING OFFSET
90EF:20 28 F8   218         JSR  VLINE       ;DRAW IT
90F2:60         219         RTS              ;SEE YOU LATER
```

LISTING 19.1d "P" COMMAND HANDLER

```
90F3:           223 ;***
90F3:           224 ;***
                225 ************************************************************
90F3:           226 ;***
90F3:           227 ;*** PROCESS THE "P" COMMAND: PXY
90F3:           228 ;***
90F3:           229 ;*** ARGUMENTS ARE - X = COLUMN OFFSET
90F3:           230 ;***                 Y = ROW OFFSET
90F3:           231 ;***
                232 ************************************************************
90F3:           233 ;***
90F3:           234 ;***


90F3:20 A0 90   236 PLT     JSR  GETARG
90F6:85 FA      237         STA  CX
90F8:20 A0 90   238         JSR  GETARG
90FB:85 F9      239         STA  RX
90FD:A5 FE      240 PLTIND  LDA  COL         ;GET CURRENT COL#
90FF:18         241         CLC              ;NORMALIZE
9100:65 FA      242         ADC  CX          ;ADD IN COLUMN OFFSET
9102:A8         243         TAY              ;PREPARE TO CALL PLOT
9103:A5 FD      244         LDA  ROW         ;GET CURRENT ROW#
9105:65 F9      245         ADC  RX          ;ADD ROW OFFSET
9107:20 00 F8   246         JSR  PLOT        ;PLOT THE DUMB THING
910A:60         247         RTS              ;SAYONARA
```

LISTING 19.1e "C" COMMAND HANDLER

```
910B:            251 ;***
910B:            252 ;***
                 253 ************************************************************
910B:            254 ;***
910B:            255 ;*** PROCESS THE "C" COMMAND: CX
910B:            256 ;***
910B:            257 ;*** ARGUMENT IS - X = COLOR VALUE TO SET
910B:            258 ;***
                 259 ************************************************************
910B:            260 ;***
910B:            261 ;***


910B:20 A0 90    263 COLR    JSR  GETARG     ;GET COLOR VALUE
910E:20 64 F8    264         JSR  SETCOL     ;GET MONITOR TO SET IT
9111:60          265         RTS             ;SIMPLE - EH WHAT??
```

LISTING 19.1f "T" COMMAND HANDLER

```
9112:            269 ;***
9112:            270 ;***
                 271 ************************************************************
9112:            272 ;***
9112:            273 ;*** PROCESS THE "T" COMMAND: TXY
9112:            274 ;***
9112:            275 ;*** ARGUMENTS ARE - X = AMOUNT TO INCREASE ROW VALUE
9112:            276 ;***                 Y = AMOUNT TO INCREASE COL VALUE
9112:            277 ;***
                 278 ************************************************************
9112:            279 ;***
9112:            280 ;***


9112:20 A0 90    282 TRNSLTE JSR  GETARG
9115:18          283         CLC
9116:65 FD       284         ADC  ROW        ;PERFORM ROW TRANSLATION
9118:85 FD       285         STA  ROW        ;SAVE NEW VALUE
911A:20 A0 90    286         JSR  GETARG     ;GET COL TRANSLATION VALUE
911D:18          287         CLC
911E:65 FE       288         ADC  COL        ;PERFORM COL TRANSLATION
9120:85 FE       289         STA  COL        ;SAVE NEW VALUE
9122:60          290         RTS             ;HIGHTAIL IT BACK TO THE RANCH
```

LISTING 19.1g "S" COMMAND HANDLER

```
9123:            294 ;***
9123:            295 ;***
                 296 ************************************************************
9123:            297 ;***
9123:            298 ;*** PROCESS THE "S" COMMAND: SXY
9123:            299 ;***
9123:            300 ;*** ARGUMENTS ARE - X = AMOUNT TO DECREASE ROW VALUE
9123:            301 ;***                 Y = AMOUNT TO DECREASE COL VALUE
9123:            302 ;***
                 303 ************************************************************
9123:            304 ;***
9123:            305 ;***


9123:20 A0 90    307 SLIDE   JSR  GETARG
9126:38          308         SEC
9127:E5 FD       309         SBC  ROW
9129:85 FD       310         STA  ROW
912B:20 A0 90    311         JSR  GETARG
912E:38          312         SEC
912F:E5 FE       313         SBC  COL
9131:85 FE       314         STA  COL
9133:60          315         RTS             ;SLIP-SLIDE ON BACK
```

LISTING 19.1h "D" COMMAND HANDLER

```
9134:            319 ;***
9134:            320 ;***
                 321 ************************************************************
9134:            322 ;***
9134:            323 ;*** PROCESS THE "D" COMMAND: DXYZ
9134:            324 ;***
9134:            325 ;*** ARGUMENTS ARE - X = STARTING ROW OFFSET
9134:            326 ;***                 Y = STARTING COL OFFSET
9134:            327 ;***                 Z = LENGTH OF DIAGONAL
9134:            328 ;***
                 329 ************************************************************
9134:            330 ;***
9134:            331 ;***


9134:20 A0 90    333 DIAGON  JSR  GETARG
9137:85 F9       334         STA  RX
9139:20 A0 90    335         JSR  GETARG
913C:85 FA       336         STA  CX
913E:20 A0 90    337         JSR  GETARG
9141:85 FB       338         STA  AX
9143:20 FD 90    339 DLOOP   JSR  PLTIND     ;USE PLOT DRIVER TO PLOT A POINT
9146:C6 FB       340         DEC  AX         ;COUNT DOWN
9148:F0 07       341         BEQ  DRTS       ;QUIT IF ALL POINTS PLOTTED
914A:E6 F9       342         INC  RX         ;INCREASE ROW OFFSET BY 1
914C:E6 FA       343         INC  CX         ;INCREASE COL OFFSET BY 1
914E:4C 43 91    344         JMP  DLOOP      ;DO NEXT POINT
9151:60          345 DRTS    RTS             ;CIAO
```

LISTING 19.1i "F" COMMAND HANDLER

```
9152:            349 ;***
9152:            350 ;***
                 351 ************************************************************
9152:            352 ;***
9152:            353 ;*** PROCESS THE "F" COMMAND: F
9152:            354 ;***
                 355 ************************************************************
9152:            356 ;***
9152:            357 ;***


9152:A5 FD       359 FILL    LDA  ROW        ;GET CURRENT ROW#
9154:48          360         PHA             ;SAVE IT ON STACK
9155:A5 FE       361         LDA  COL        ;GET CURRENT COL#
9157:48          362         PHA             ;SAVE IT ON STACK
9158:A9 00       363         LDA  #$00
915A:85 FD       364         STA  ROW        ;SET ROW# = 0
915C:85 FE       365         STA  COL        ;SET COL# = 0
915E:85 F9       366         STA  RX         ;SET ROW OFFSET = 0
9160:85 FB       367         STA  AX         ;SET COL OFFSET = 0
9162:A9 27       368         LDA  #$27
9164:85 FA       369         STA  CX         ;SET WIDTH TO 39
9166:A2 30       370         LDX  #$30       ;PREPARE TO PLOT 48 LINES
9168:20 BC 90    371 FLOOP   JSR  HRZIND     ;PLOT ONE
916B:E6 FD       372         INC  ROW        ;MOVE TO NEXT
916D:CA          373         DEX             ;COUNT DOWN
916E:D0 F8       374         BNE  FLOOP      ;MORE IF NOT ZERO
9170:68          375         PLA
9171:85 FE       376         STA  COL        ;RESTORE OLD COL#
9173:68          377         PLA
9174:85 FD       378         STA  ROW        ;RESTORE OLD ROW#
9176:60          379         RTS             ;WE'VE HAD OUR FILL!
```

LISTING 19.1j "B" COMMAND HANDLER

```
9177:            383 ;***
9177:            384 ;***
                 385 ************************************************************
9177:            386 ;***
9177:            387 ;*** PROCESS THE "B" COMMAND: BXYZW
9177:            388 ;***
9177:            389 ;*** ARGUMENTS ARE - X = ROW OFFSET TO START BLOCK
9177:            390 ;***                Y = COL OFFSET TO START BLOCK
9177:            391 ;***                Z = HEIGHT OF BLOCK (# OF ROWS)
9177:            392 ;***                W = WIDTH OF BLOCK (HLIN SIZE)
9177:            393 ;***
                 394 ************************************************************
9177:            395 ;***
9177:            396 ;***


9177:20 A0 90    398 BLOCK   JSR  GETARG
917A:85 F9       399         STA  RX
917C:20 A0 90    400         JSR  GETARG
```

LISTING 19.1 *(cont.)*

```
917F:85 FA      401           STA  CX
9181:20 A0 90   402           JSR  GETARG
9184:85 FB      403           STA  AX
9186:20 A0 90   404           JSR  GETARG
9189:85 FC      405           STA  BX
918B:C6 FC      406           DEC  BX          ;ZERO-INDEX ADJUSTMENT
918D:18         407 BLOOP     CLC
918E:A5 FE      408           LDA  COL         ;GET CURRENT COL#
9190:65 FA      409           ADC  CX          ;ADD INITIAL COL OFFSET
9192:A8         410           TAY              ;SAVE IN Y-REG
9193:65 FC      411           ADC  BX          ;ADD LINE WIDTH
9195:85 2C      412           STA  H2          ;SAVE IN ENDING COLUMN VALUE
9197:A5 FD      413           LDA  ROW         ;GET CURRENT ROW#
9199:65 F9      414           ADC  RX          ;ADD IN ROW OFFSET
919B:20 19 F8   415           JSR  HLINE       ;PLOT THE LINE
919E:E6 F9      416           INC  RX          ;MOVE TO NEXT ROW
91A0:C6 FB      417           DEC  AX          ;COUNT DOWN # OF ROWS TO PLOT
91A2:D0 E9      418           BNE  BLOOP       ;DONE WHEN WE GET TO 0
91A4:60         419           RTS              ;BID A FOND ADIEU
91A5:           420 ;***
91A5:           421 ;***
------ NEXT OBJECT FILE NAME IS LISTING 19.1 *****.OBJ1
03F5:           422           ORG  $3F5
03F5:4C 00 90   423           JMP  START

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

SYMBOL TABLE SORTED BY SYMBOL

```
908B AQUIT      9092 ARTS       FB AX       9177 BLOCK
918D BLOOP        FC BX           08 CHARADDR   FF CHARS
  B1 CHRGET       B7 CHRGOT   910B COLR        FE COL
  FA CX         9028 DEC1     9032 DEC2     903C DEC3
9046 DEC4       9050 DEC5     905A DEC6     9064 DEC7
906E DEC8       9019 DECODE     06 DESC     9134 DIAGON
9143 DLOOP      907E DONE     9081 DONE1    9151 DRTS
  D0 ERRFLG     9078 ERR      D412 ERROR    9152 FILL
9168 FLOOP      90AC GA1      909D GCRTS    90A0 GETARG
9093 GETCHAR      2C H2       F819 HLINE    90BC HRZIND
90AD HRZNTL     F800 PLOT     90FD PLTIND   90F3 PLT
DFE3 PTRGET       FD ROW        F9 RX       F864 SETCOL
9123 SLIDE      9000 START    9112 TRNSLTE    2D V2
F828 VLINE      90D0 VRTCL
```

SYMBOL TABLE SORTED BY ADDRESS

```
  06 DESC          08 CHARADDR      2C H2            2D V2
  B1 CHRGET        B7 CHRGOT        D0 ERRFLG        F9 RX
  FA CX            FB AX            FC BX            FD ROW
  FE COL           FF CHARS       9000 START       9019 DECODE
9028 DEC1        9032 DEC2        903C DEC3        9046 DEC4
9050 DEC5        905A DEC6        9064 DEC7        906E DEC8
9078 ERR         907E DONE        9081 DONE1       908B AQUIT
9092 ARTS        9093 GETCHAR     909D GCRTS       90A0 GETARG
90AC GA1         90AD HRZNTL      90BC HRZIND      90D0 VRTCL
90F3 PLT         90FD PLTIND      910B COLR        9112 TRNSLTE
9123 SLIDE       9134 DIAGON      9143 DLOOP       9151 DRTS
9152 FILL        9168 FLOOP       9177 BLOCK       918D BLOOP
D412 ERROR       DFE3 PTRGET      F800 PLOT        F819 HLINE
F828 VLINE       F864 SETCOL
```

LISTING 19.2a PRINT OVER PERFORATIONS—EQUATES

```
0000:              2          MSB   OFF
0006:              3 CRCOUNT  EQU   $6        ;COUNTS CARRIAGE RETURNS ON A PAGE
0007:              4 MSGPTR   EQU   $7
000A:              5 MAXMSG   EQU   $0A
FDF0:              6 COUT1    EQU   $FDF0     ;MONITOR CHARACTER OUTPUT ROUTINE
0036:              7 CSWL     EQU   $36
0037:              8 CSWH     EQU   $37
008C:              9 CTRLL    EQU   $8C
008D:             10 CTRLM    EQU   $8D
00B1:             11 CHRGET   EQU   $B1
03EA:             12 MVSW     EQU   $3EA      ;ROUTINE WHICH RECONNECTS DOS
C100:             13 INIT     EQU   $C100     ;PRINTER INIT ROUTINE
C102:             14 PRINT    EQU   $C102     ;ADDRESS OF PRINTER FIRMWARE ROUTINE
FE93:             15 SETVID   EQU   $FE93
----- NEXT OBJECT FILE NAME IS LISTING 19.2 *****.OBJ0
9000:             16          ORG   $9000
```

LISTING 19.2b MAIN DRIVER

```
9000:             20 ;***
9000:             21 ;***
                  22 ************************************************************
9000:             23 ;***
9000:             24 ;***     A  M  P  E  R   -   L  I  S  T
9000:             25 ;***
9000:             26 ;***  THESE ROUTINES ARE MEANT TO BE INVOKED FROM
9000:             27 ;*** APPLESOFT IN DIRECT COMMAND MODE.  THE
9000:             28 ;*** OPTIONS AVAILABLE ARE AS FOLLOWS:
9000:             29 ;***
9000:             30 ;***      &I    -    INITIALIZE: TURN ON PRINTER AND
9000:             31 ;***                  PUT THE PRINT OVER PERFORATIONS
9000:             32 ;***                  ROUTINE INTO THE OUTPUT HOOKS.
9000:             33 ;***
9000:             34 ;***      &U    -    UNHOOK: REMOVE THE PRINT OVER
9000:             35 ;***                  PERF ROUTINE FROM THE OUTPUT
9000:             36 ;***                  HOOKS.
9000:             37 ;***
9000:             38 ;***      &F    -    ISSUE A FORM FEED TO THE
9000:             39 ;***                  PRINTER AND RESET THE LINE
9000:             40 ;***                  AND PAGE COUNT.
9000:             41 ;***
9000:             42 ;***      &"    -    ACCEPT TITLE STRING.  TITLE
9000:             43 ;***                  IS ENDED BY A ". THE TITLE
9000:             44 ;***                  THUS SPECIFIED WILL BE
9000:             45 ;***                  PRINTED ON EVERY PAGE OF
9000:             46 ;***                  LISTING EXCEPT THE FIRST.
9000:             47 ;***
9000:             48 ;***
                  49 ************************************************************
9000:             50 ;***
9000:             51 ;***


9000:C9 49        53 ALIST    CMP  #'I        ;IS IT INIT COMMAND?
9002:D0 06        54          BNE  ST1        ;NOPE
9004:20 31 90     55          JSR  INITPRT    ;YES - DO INITIALIZATIONS
9007:4C 2B 90     56          JMP  DONE
900A:C9 55        57 ST1      CMP  #'U        ;IS IT UNHOOK COMMAND?
900C:D0 06        58          BNE  ST2        ;NOPE
900E:20 5D 90     59          JSR  UNHOOK     ;YES - GET US OFF THE HOOK
9011:4C 2B 90     60          JMP  DONE
9014:C9 46        61 ST2      CMP  #'F        ;IS IT FORM FEED COMMAND?
9016:D0 06        62          BNE  ST3        ;NOPE
9018:20 51 92     63          JSR  PRFF       ;YES - ISSUE FORM FEED, ETC.
901B:4C 2B 90     64          JMP  DONE
901E:C9 22        65 ST3      CMP  #'"        ;IS IT TITLE COMMAND?
9020:D0 06        66          BNE  ST4        ;NOPE
9022:20 9A 91     67          JSR  TITLE      ;YES - COLLECT TITLE
9025:4C 2B 90     68          JMP  DONE
9028:20 64 90     69 ST4      JSR  HELPMESS   ;THE DUMMY NEEDS HELP
902B:20 B1 00     70 DONE     JSR  CHRGET     ;FIND THE END OF THE LINE
902E:D0 FB        71          BNE  DONE
9030:60           72          RTS
```

LISTING 19.2c HOOK SET AND UNSET ROUTINES

```
9031:                76 ;***
9031:                77 ;***
                     78 ************************************************************
9031:                79 ;***
9031:                80 ;*** ROUTINE TO CONNECT US INTO OUTPUT HOOK
9031:                81 ;***
                     82 ************************************************************
9031:                83 ;***
9031:                84 ;***


9031:20 00 C1        86 INITPRT JSR  INIT        ;TURN ON PRINTER
9034:A9 65           87         LDA  #>PERF      ;LOW BYTE OF ADDRESS
9036:85 36           88         STA  CSWL
9038:A9 92           89         LDA  #<PERF      ;HIGH BYTE OF ADDRESS
903A:85 37           90         STA  CSWH
903C:A9 00           91         LDA  #$00
903E:85 06           92         STA  CRCOUNT     ;INIT LINE COUNT TO 0
9040:A9 20           93         LDA  #$20
9042:8D F7 91        94         STA  PGNUM
9045:A9 31           95         LDA  #$31
9047:8D F8 91        96         STA  PGNUM+1
904A:A2 3F           97         LDX  #$3F
904C:A9 20           98         LDA  #$20
904E:9D B1 91        99 TBLANK  STA  TICHARS,X
9051:CA             100         DEX
9052:D0 FA          101         BNE  TBLANK
9054:A9 22          102         LDA  #$22
9056:8D B1 91       103         STA  TICHARS
9059:20 EA 03       104         JSR  MVSW        ;TELL DOS OF OUR INTENTIONS
905C:60             105         RTS




905D:               107 ;***
905D:               108 ;***
                    109 ************************************************************
905D:               110 ;***
905D:               111 ;*** THIS UNHOOKS OUR ROUTINE
905D:               112 ;***
                    113 ************************************************************
905D:               114 ;***
905D:               115 ;***


905D:20 93 FE       117 UNHOOK JSR  SETVID       ;SIMULATE PR#0
9060:20 EA 03       118        JSR  MVSW         ;RECONNECT DOS
9063:60             119        RTS
```

LISTING 19.2d HELP MESSAGES

```
9064:           123 ;***
9064:           124 ;***
                125 ************************************************************
9064:           126 ;***
9064:           127 ;***    HELPMESS ROUTINE
9064:           128 ;***
9064:           129 ;***  PRINT A LIST OF LEGAL COMMANDS TO
9064:           130 ;***  THE SCREEN.  USER HAS ENTERED AN ILLEGAL
9064:           131 ;***  COMMAND.
9064:           132 ;***
                133 ************************************************************
9064:           134 ;***
9064:           135 ;***


9064:20 A7 90   137 HELPMESS JSR BLANK      ;BLANK LINE TO SCREEN
9067:20 A7 90   138          JSR  BLANK
906A:A2 00      139          LDX  #$00
906C:E0 0A      140 HLPLOOP  CPX  #MAXMSG
906E:B0 13      141          BCS  HRTS
9070:8A         142          TXA
9071:48         143          PHA             ;SAVE X VALUE
9072:2A         144          ROL  A          ;CONVERT X VALUE TO INDEX
9073:AA         145          TAX
9074:20 84 90   146          JSR  GETPTR
9077:20 90 90   147          JSR  OUTTXT     ;PRINT MESSAGE LINE
907A:20 A7 90   148          JSR  BLANK      ;NEWLINE ON SCREEN
907D:68         149          PLA             ;RETRIEVE UNDOUBLED X VALUE
907E:AA         150          TAX             ;PUT BACK IN X
907F:E8         151          INX
9080:4C 6C 90   152          JMP  HLPLOOP
9083:60         153 HRTS     RTS             ;NO MORE MESSAGES


9084:           155 ;***
9084:           156 ;***
                157 ************************************************************
9084:           158 ;***
9084:           159 ;***  GETPTR ROUTINE
9084:           160 ;***
9084:           161 ;***  PICK MESSAGE ADDRESS OUT OF MSGTAB.  USES
9084:           162 ;***  CONTENTS OF X-REG AS INDEX.
9084:           163 ;***
                164 ************************************************************
9084:           165 ;***
9084:           166 ;***


9084:BD AD 90   168 GETPTR  LDA  MSGTAB,X
9087:85 07      169          STA  MSGPTR
9089:E8         170          INX
908A:BD AD 90   171          LDA  MSGTAB,X
908D:85 08      172          STA  MSGPTR+1
908F:60         173          RTS
```

```
9090:           175 ;***
9090:           176 ;***
                177 ************************************************************
9090:           178 ;***
9090:           179 ;***  OUTTXT ROUTINE
9090:           180 ;***
9090:           181 ;***  OUTPUT TEXT POINTED TO BY MSGPTR
9090:           182 ;***  NUL MARKS THE END OF THE MESSAGE
9090:           183 ;***
                184 ************************************************************
9090:           185 ;***
9090:           186 ;***


9090:98         188 OUTTXT  TYA             ;SAVE Y-REG
9091:48         189         PHA             ;ON 6502 STACK
9092:A0 00      190         LDY  #$00       ;USE INDIRECT ADDRESSING
9094:B1 07      191 OLOOP   LDA  (MSGPTR),Y
9096:F0 0C      192         BEQ  OQUIT      ;TEXT ENDS WITH A NUL
9098:20 F0 FD   193         JSR  COUT1      ;SEND IT TO SCREEN
909B:E6 07      194         INC  MSGPTR     ;BUMP PTR
909D:D0 F5      195         BNE  OLOOP
909F:E6 08      196         INC  MSGPTR+1
90A1:4C 94 90   197         JMP  OLOOP
90A4:68         198 OQUIT   PLA             ;GET OLD Y VALUE BACK
90A5:A8         199         TAY             ;FROM STACK AND PUT IT IN Y
90A6:60         200         RTS


90A7:           202 ;***
90A7:           203 ;***
                204 ************************************************************
90A7:           205 ;***
90A7:           206 ;***  BLANK - SEND CR TO SCREEN
90A7:           207 ;***
                208 ************************************************************
90A7:           209 ;***
90A7:           210 ;***


90A7:A9 8D      212 BLANK   LDA  #CTRLM
90A9:20 F0 FD   213         JSR  COUT1
90AC:60         214         RTS


90AD:           216 ;***
90AD:           217 ;***
                218 ************************************************************
90AD:           219 ;***
90AD:           220 ;***      M S G T A B
90AD:           221 ;***
90AD:           222 ;*** MESSAGE ADDRESSES
90AD:           223 ;***
                224 ************************************************************
90AD:           225 ;***
90AD:           226 ;***
```

LISTING 19.2 (*cont.*)

```
90AD:C1 90      228 MSGTAB  DW    MSG1
90AF:E0 90      229         DW    MSG2
90B1:FF 90      230         DW    MSG3
90B3:E0 90      231         DW    MSG2
90B5:1E 91      232         DW    MSG4
90B7:3D 91      233         DW    MSG5
90B9:5C 91      234         DW    MSG6
90BB:7B 91      235         DW    MSG7
90BD:E0 90      236         DW    MSG2
90BF:C1 90      237         DW    MSG1


90C1:           239 ;***
90C1:           240 ;***
                241 ************************************************************
90C1:           242 ;***
90C1:           243 ;*** MESSAGE TEXTS
90C1:           244 ;***
                245 ************************************************************
90C1:           246 ;***
90C1:           247 ;***


90C1:           249         MSB  ON
90C1:AA AA AA   250 MSG1    ASC/**********************************/
90C4:AA AA AA
90C7:AA AA AA
90CA:AA AA AA
90CD:AA AA AA
90D0:AA AA AA
90D3:AA AA AA
90D6:AA AA AA
90D9:AA AA AA
90DC:AA AA AA
90DF:00         251         DFB  0
90E0:AA A0 A0   252 MSG2    ASC/*                                      */
90E3:A0 A0 A0
90E6:A0 A0 A0
90E9:A0 A0 A0
90EC:A0 A0 A0
90EF:A0 A0 A0
90F2:A0 A0 A0
90F5:A0 A0 A0
90F8:A0 A0 A0
90FB:A0 A0 AA
90FE:00         253         DFB  0
90FF:AA A0 CC   254 MSG3    ASC/* LEGAL     COMMANDS ARE:          */
9102:C5 C7 C1
9105:CC A0 C3
9108:CF CD CD
910B:C1 CE C4
910E:D3 A0 C1
```

```
9111:D2 C5 BA
9114:A0 A0 A0
9117:A0 A0 A0
911A:A0 A0 AA
911D:00          255           DFB  0
911E:AA A0 A0    256 MSG4      ASC/*           &I - INITIALIZE              */
9121:A6 C9 A0
9124:AD A0 C9
9127:CE C9 D4
912A:C9 C1 CC
912D:C9 DA C5
9130:A0 A0 A0
9133:A0 A0 A0
9136:A0 A0 A0
9139:A0 A0 AA
913C:00          257           DFB  0
913D:AA A0 A0    258 MSG5      ASC/*           &U - UNHOOK                  */
9140:A6 D5 A0
9143:AD A0 D5
9146:CE C8 CF
9149:CF CB A0
914C:A0 A0 A0
914F:A0 A0 A0
9152:A0 A0 A0
9155:A0 A0 A0
9158:A0 A0 AA
915B:00          259           DFB  0
915C:AA A0 A0    260 MSG6      ASC/*           &F - NEW PAGE                */
915F:A6 C6 A0
9162:AD A0 CE
9165:C5 D7 A0
9168:D0 C1 C7
916B:C5 A0 A0
916E:A0 A0 A0
9171:A0 A0 A0
9174:A0 A0 A0
9177:A0 A0 AA
917A:00          261           DFB  0
917B:AA A0 A0    262 MSG7      ASC/*           &"TITLE" - SETS UP TITLE     */
917E:A6 A2 D4
9181:C9 D4 CC
9184:C5 A2 A0
9187:AD A0 D3
918A:C5 D4 D3
918D:A0 D5 D0
9190:A0 D4 C9
9193:D4 CC C5
9196:A0 A0 AA
9199:00          263           DFB  0
919A:            264           MSB  OFF
```

LISTING 19.2e TITLE ROUTINES

```
919A:            268 ;***
919A:            269 ;***
                 270 ************************************************************
919A:            271 ;***
919A:            272 ;***  TITLE ROUTINE
919A:            273 ;***
919A:            274 ;*** COLLECT TITLE CHARACTERS AND PUT THEM
919A:            275 ;*** INTO A BUFFER.
919A:            276 ;***
                 277 ************************************************************
919A:            278 ;***
919A:            279 ;***


919A:A2 00       281 TITLE   LDX  #$00
919C:20 B1 00    282 TLOOP   JSR  CHRGET
919F:C9 22       283         CMP  #'"
91A1:F0 0D       284         BEQ  TRTS
91A3:C9 2D       285         CMP  #'-
91A5:D0 02       286         BNE  STORE
91A7:A9 20       287         LDA  #$20       ;REPLACE "-" BY BLANK
91A9:9D B1 91    288 STORE   STA  TICHARS,X
91AC:E8          289         INX
91AD:4C 9C 91    290         JMP  TLOOP
91B0:60          291 TRTS    RTS
91B1:22          292 TICHARS DFB  '"          ;INIT TITLE TO EMPTY STRING
91B2:            293         DS   63
91F1:50 41 47    294         ASC  /PAGE      "/
91F4:45 20 22
91F7:20 30 22    295 PGNUM   ASC  /          0"/


91FA:            297 ;***
91FA:            298 ;***
                 299 ************************************************************
91FA:            300 ;***
91FA:            301 ;***      PRINT TITLE
91FA:            302 ;***
91FA:            303 ;***  PRINT TITLE AT THE TOP OF NEW PAGE.
91FA:            304 ;*** FOLLOW IT BY A BLANK LINE FOR READABILITY.
91FA:            305 ;***
                 306 ************************************************************
91FA:            307 ;***
91FA:            308 ;***


91FA:A2 00       310 PRTITLE LDX  #$00
91FC:BD B1 91    311 PRLOOP  LDA  TICHARS,X
91FF:E8          312         INX
9200:C9 22       313         CMP  #'"
9202:F0 06       314         BEQ  PRFIN
9204:20 02 C1    315         JSR  PRINT
9207:4C FC 91    316         JMP  PRLOOP
```

```
920A:20 2B 92   317 PRFIN   JSR  INCPGNUM  ;BUMP PAGE NUMBER
920D:20 1A 92   318         JSR  PRTPGNUM  ;PRINT PAGE NUMBER
9210:20 4B 92   319         JSR  PRCR      ;END TITLE LINE
9213:20 4B 92   320         JSR  PRCR       ;PUT IN SPACING LINE
9216:20 4B 92   321         JSR  PRCR      ;AND ANOTHER
9219:60         322         RTS
921A:           326 ;***
921A:           327 ;***
                328 ************************************************************
921A:           329 ;***
921A:           330 ;***  PRINT THE PAGE NUMBER ON TITLE LINE
921A:           331 ;***
                332 ************************************************************
921A:           333 ;***
921A:           334 ;***


921A:A2 00      336 PRTPGNUM LDX #$00
921C:BD F7 91   337 PGLOOP  LDA  PGNUM,X
921F:C9 22      338         CMP  #'"
9221:F0 07      339         BEQ  PG2
9223:20 02 C1   340         JSR  PRINT
9226:E8         341         INX
9227:4C 1C 92   342         JMP  PGLOOP
922A:60         343 PG2     RTS


922B:           345 ;***
922B:           346 ;***
                347 ************************************************************
922B:           348 ;***
922B:           349 ;***  INCREMENT THE PAGE NUMBER
922B:           350 ;***  WE HANDLE UP TO 99.
922B:           351 ;***
                352 ************************************************************
922B:           353 ;***
922B:           354 ;***


922B:EE F8 91   356 INCPGNUM INC PGNUM+1   ;BUMP UNITS DIGIT
922E:AD F8 91   357         LDA  PGNUM+1   ;CHECK IT
9231:C9 39      358         CMP  #$39      ;IS IT > 9?
9233:90 15      359         BCC  PGRTS     ;NO - GO BACK
9235:A9 30      360         LDA  #$30      ;YES - CHANGE TO 0
9237:8D F8 91   361         STA  PGNUM+1   ;SAVE IT
923A:AD F7 91   362         LDA  PGNUM     ;NOW TENS DIGIT
923D:C9 20      363         CMP  #$20      ;IF = " ",THEN MAKE IT 1
923F:F0 04      364         BEQ  FIRST
9241:EE F7 91   365         INC  PGNUM     ;OTHERWISE - BUMP
9244:60         366         RTS            ;AND RUN!!
9245:A9 31      367 FIRST   LDA  #$31      ;MAKE IT A 1
9247:8D F7 91   368         STA  PGNUM     ;SAVE IN TENS DIGIT
924A:60         369 PGRTS   RTS
```

LISTING 19.2f MISC ROUTINES

```
924B:            373 ;***
924B:            374 ;***
                 375 ************************************************************
924B:            376 ;***
924B:            377 ;***  PRINT A CARRIAGE RETURN
924B:            378 ;***
                 379 ************************************************************
924B:            380 ;***
924B:            381 ;***


924B:A9 8D       383 PRCR    LDA  #CTRLM
924D:20 02 C1    384         JSR  PRINT
9250:60          385         RTS


9251:            387 ;***
9251:            388 ;***
                 389 ************************************************************
9251:            390 ;***
9251:            391 ;***  PRINT A FORM FEED AND RESET COUNTERS
9251:            392 ;***
                 393 ************************************************************
9251:            394 ;***
9251:            395 ;***


9251:A9 8C       397 PRFF    LDA  #CTRLL
9253:20 02 C1    398         JSR  PRINT
9256:A9 00       399         LDA  #$00
9258:85 06       400         STA  CRCOUNT
925A:A9 20       401         LDA  #$20
925C:8D F7 91    402         STA  PGNUM
925F:A9 31       403         LDA  #$31
9261:8D F8 91    404         STA  PGNUM+1
9264:60          405         RTS
```

LISTING 19.2g PERF—HOOK ROUTINE

```
9265:            409 ;***
9265:            410 ;***
                 411 ************************************************************
9265:            412 ;***
9265:            413 ;***         P   E   R   F
9265:            414 ;***
9265:            415 ;*** THIS ROUTINE COUNTS LINES OUTPUT TO THE EPSON
9265:            416 ;*** PRINTER AND INSERTS A FORM FEED AFTER EVERY
9265:            417 ;*** 58 LINES.  THE ROUTINE IS USED WHEN LISTING
9265:            418 ;*** BASIC PROGRAMS IN ORDER TO FORMAT THEM
9265:            419 ;*** NICELY, AVOIDING THE PRINTING ON TOP OF
9265:            420 ;*** PAGE PERFORATIONS.
9265:            421 ;***
                 422 ************************************************************
9265:            423 ;***
9265:            424 ;***
```

```
9265:C9 8D      426 PERF    CMP  #CTRLM    ;CHECK FOR CARRIAGE RETURN
9267:F0 08      427         BEQ  COUNTIT
9269:C9 8C      428         CMP  #CTRLL    ;CHECK FOR FORM-FEED
926B:F0 10      429         BEQ  FEED      ;NEW PAGE IF SO
926D:20 02 C1   430         JSR  PRINT     ;NOT CR - JUST DO OUTPUT
9270:60         431         RTS
9271:E6 06      432 COUNTIT INC  CRCOUNT   ;ADD ONE TO LINE COUNT
9273:A5 06      433         LDA  CRCOUNT   ;NOW CHECK IT
9275:C9 3A      434         CMP  #$3A      ;58 IS THE MAGIC NUMBER
9277:B0 04      435         BCS  FEED      ;IF >= 58 THEN THROW PAGE
9279:20 4B 92   436         JSR  PRCR      ;OTHERWISE, JUST PRINT CR
927C:60         437         RTS
927D:20 4B 92   438 FEED    JSR  PRCR      ;PUT IN CR
9280:A9 8C      439         LDA  #CTRLL    ;NOW FORM FEED PRINTER
9282:20 02 C1   440         JSR  PRINT
9285:A9 04      441         LDA  #$04
9287:85 06      442         STA  CRCOUNT   ;RESET COUNT
9289:20 FA 91   443         JSR  PRTITLE   ;PRINT TITLE LINES
928C:60         444         RTS

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

SYMBOL TABLE SORTED BY SYMBOL

```
?9000 ALIST      90A7 BLANK      00B1 CHRGET     9271 COUNTIT
FDF0 COUT1       0006 CRCOUNT    0037 CSWH       0036 CSWL
008C CTRLL       008D CTRLM      902B DONE       927D FEED
9245 FIRST       9084 GETPTR     9064 HELPMESS   906C HLPLOOP
9083 HRTS        922B INCPGNUM   C100 INIT       9031 INITPRT
000A MAXMSG      90C1 MSG1       90E0 MSG2       90FF MSG3
911E MSG4        913D MSG5       915C MSG6       917B MSG7
0007 MSGPTR      90AD MSGTAB     03EA MVSW       9094 OLOOP
90A4 OQUIT       9090 OUTTXT     9265 PERF       922A PG2
921C PGLOOP      91F7 PGNUM      924A PGRTS      924B PRCR
9251 PRFF        920A PRFIN      C102 PRINT      91FC PRLOOP
91FA PRTITLE     921A PRTPGNUM   FE93 SETVID     900A ST1
9014 ST2         901E ST3        9028 ST4        91A9 STORE
904E TBLANK      91B1 TICHARS    919A TITLE      919C TLOOP
91B0 TRTS        905D UNHOOK
```

LISTING 19.2 (*cont.*)

SYMBOL TABLE SORTED BY ADDRESS

```
  06 CRCOUNT          07 MSGPTR          0A MAXMSG          36 CSWL
  37 CSWH             8C CTRLL           8D CTRLM           B1 CHRGET
03EA MVSW           ?9000 ALIST        900A ST1           9014 ST2
901E ST3            9028 ST4           902B DONE          9031 INITPRT
904E TBLANK         905D UNHOOK        9064 HELPMESS      906C HLPLOOP
9083 HRTS           9084 GETPTR        9090 OUTTXT        9094 OLOOP
90A4 OQUIT          90A7 BLANK         90AD MSGTAB        90C1 MSG1
90E0 MSG2           90FF MSG3          911E MSG4          913D MSG5
915C MSG6           917B MSG7          919A TITLE         919C TLOOP
91A9 STORE          91B0 TRTS          91B1 TICHARS       91F7 PGNUM
91FA PRTITLE        91FC PRLOOP        920A PRFIN         921A PRTPGNUM
921C PGLOOP         922A PG2           922B INCPGNUM      9245 FIRST
924A PGRTS          924B PRCR          9251 PRFF          9265 PERF
9271 COUNTIT        927D FEED          C100 INIT          C102 PRINT
FDF0 COUT1          FE93 SETVID
```

LISTING 19.3 AMPER LETTERS LISTER

```
]LIST

1  REM  ============================
2  REM  =                          =
3  REM  =      AMPER LETTERS       =
4  REM  =                          =
5  REM  =DR. RICHARD C. VILE, JR.=
6  REM  = ALL COMMERCIAL RIGHTS  =
7  REM  =        RESERVED          =
8  REM  =                          =
9  REM  ============================
10 D$ =  CHR$ (4): REM CONTROL-D
15  DIM L$(94): REM  LETTERS ARRAY
20 F$ = "GIANT LETTERS DATA"
25  PRINT D$;"OPEN ";F$
26  PRINT D$;"READ ";F$
28  FOR I = 1 TO 94: INPUT L$(I): NEXT I
30  PRINT D$;"CLOSE ";F$
99  POKE 208,0: POKE 216,0
100  & I
105  & "***--AMPER-LETTERS-PATTERNS--***"
110  & F
112  PRINT "***  AMPER LETTERS PATTERNS  ***": PRINT : PRINT
115  FOR I = 1 TO 94
116  IF I < 10 THEN  PRINT " ";
117  PRINT I;"     ";L$(I)
120  NEXT I
140  & F
145  & U
199  END

]
```

Chapter

20

Dazzling Programs

Many impressive and entertaining graphics displays may be generated using only the simple features of the APPLE II low-resolution display. Using BASIC may cause the speed of the display generation to be too slow, however. This is where machine language naturally comes into its own. In this chapter we present some dazzling display programs using 6502 assembler.

1. FOLLOW THE BOUNCING BALL

By now, virtually everyone has played some form of video ping-pong. A small blob moves around the display and the players bat at it with their electronic paddles. The "ball" somehow knows how to rebound from the paddles and even how to reflect at the proper angle when it hits a wall.

The pong games contain the seed of an idea for producing graphics displays. Suppose the ball left its imprint at every location it visited? Then the screen would gradually be filled with lines marking the ball's travel route. This would not be all that exciting if the lines remained the same color and were never erased. The screen would simply turn slowly into a solid color the same as the ball. But suppose the ball changed color periodically. Then its trail would form an abstract design which would constantly shift and change depending on where the ball was and how long it had been moving. In addition the ball could periodically *erase* parts of the design as well as plot new colors. The proportion of plotting to erasing would determine the density of the design displayed. We shall refer to programs of this general outline as "dazzlers" or to any specific one as an implementation of "dazzle."

2. APPLE DAZZLE

Listings 20.1 and 20.2 present an implementation of Dazzle. Listing 20.1 is written in Integer BASIC. It is a program which supervises the production of the Dazzle display. Listing 20.2 is a program written 6502 assembler and implements the dazzle graphics.

In this version of the program, the moving blob has the following characteristics:

- It bounces off the walls of the display.
- It changes color periodically. This is controlled by the BASIC program variable CHANGEVALUE.

The BASIC program passes CHANGEVALUE on to the machine-language program. The machine-language code plots CHANGEVALUE different positions of the blob, all in the same color, before returning. The BASIC code alternates between setting COLOR=0 and randomly choosing a color value between 1 and 15. This means that the machine-language code alternates between plotting and erasing. This gives the display its dynamic appearance. To see the effect this has, modify lines 145 and 146 of the program.

- The blob changes direction in midscreen. The change of direction may be either horizontal or vertical. This is chosen randomly and is indicated by the BASIC program variable DBNDVALUE:

DBNDVALUE = 0 → Bend in Y direction
DBNDVALUE = 1 → Bend in X direction

The bending occurs regularly, just as does the color change. The "time" between bends is controlled by the BASIC program variable BNDTIMEVALUE.

The responsibilities of the BASIC program are as follows:

- Set up user controllable parameters: in the program of Listing 20.1 there are none of these. See the Explorations section for suggestions about incorporating some.
- Set up the parameters which stay fixed for a given "generation" of the display:

Bend time how many blobs are plotted before a midscreen bend occurs?

Bend direction which direction will the midscreen bends take?

Change time how many blobs are plotted consecutively in the same color, or how many are erased before changing color?

DX,DY values how far the blob moves in the X and Y directions between each individual plot

- Repeatedly invoke the machine-language dazzler until some desired change point. In this version, that occurs when the user hits the RETURN key.

3. IMPLEMENTING DAZZLE

In this section we discuss the implementation of the programs of Listings 20.1 and 20.2.

Using pseudo-code

The code of the Dazzle program was not created *directly* in BASIC and assembly language. Rather it was first written in an informal style known as "pseudo-code." Then it was refined or translated into the final target code.

Using pseudo-code gives a programmer some interesting advantages:

- It allows programs to be designed in problem-oriented terms, rather than in machine- or language-constrained terms.
- It provides documentation for the derived program. The pseudo-code itself may be retained in the body of the program as comments. Or it may be kept, separately, as a specification of the actions taken by the program code. In any case, it helps to clarify the meaning of the final program.
- It enables assembly-language programs to first be expressed in a higher-level notation.
- It allows the programmer to ignore coding details at first.
- It allows the programmer to concentrate on the structure and design of the program, rather than the nits.

Pseudo-Code Development of Listing 20.1

The Integer BASIC program of Listing 20.1 at the end of the chapter was developed by first writing down its responsibilities. These have been given above in detail. The specification of these responsibilities was converted into a pseudo-code outline program as follows:

```
prolog (declarations)                [lines 1–99]
repeat
  Set-up user-controlled
  parameters                         [not implemented]
  Set-up "fixed" parameters          [lines 105–119]
    repeat
    Flip-flop between plot and
    erase.                           [lines 145–146]
    Local set-up                     [empty]
    Invoke machine-language
    dazzler.                         [line 150]
  until timeout or
  (keypress=RETURN)                  [lines 155–199]
until tired;
```

Comparing this version to the actual program code shows the level of refinement necessary to turn pseudo-

code into real code. The pseudo-code version clearly reveals the structure of the program. It is a good idea to consult it when making a change to the program. For example, suppose we wished to add the timeout feature for ending the inner loop (which was not included in the program of Listing 20.1). First, what do we mean by a timeout?

Timeout in this context means that the inner loop will go through a predetermined number of cycles. Then if the user does not press RETURN during that interval, the program will automatically wipe out the current pattern and start a new one.

This feature may be implemented by adding a counter variable to the code: call it TIMEOUT. The counter must be started at 0 in the outer loop. Then it must be bumped and tested in the inner loop. The changes could be incorporated with the addition of the following lines:

```
20 MAXTIME = 100
   ...
114 TIMEOUT = 0
   ...
165 TIMEOUT = TIMEOUT + 1
166 IF TIMEOUT >= MAXTIME THEN GOTO 105
```

If we make these changes, then the pseudo-code version should also be changed to reflect them. Since the initialization of the timeout counter is already implicit in the description of the setup, it is only necessary to insert a line of pseudo-code:

Bump timeout counter [line 165]

just before the line containing the first *until*.

Pseudo-Code Development of Listing 20.2

Now we show the pseudo-code for the machine-language driver program. The requirements for this program are to control the blob. The characteristics of the blob have been outlined above.

```
BEND = 0
FOR INDEX := 1 TO CHANGE DO
PLOT X,Y
X = X + DX
Y = Y + DY
IF X<0 THEN FIX1
IF X>=40 THEN FIX2
IF Y<0   THEN FIX3
IF Y>=48 THEN Y=0

BEND = BEND + 1
IF BEND >= BENDTIME
THEN
    BEND=0
    IF BENDDIRECTION = 0
    THEN
        DX = -DX
    ELSE
        DY = -DY
    ENDIF
ENDIF
ENDDO
RETURN
```

The subroutines FIX1, FIX2, and FIX3 are pseudo-coded as follows:

```
FIX1: X = 0
      DX = -DX
      RETURN
FIX2: X = 39
      DX = -DX
      RETURN
FIX3: Y = 0
      DY = -DY
      RETURN
```

Notice that if Y is greater than or equal to 48 then Y is set back to 0, but no change of Y direction is made. This means that when the blob goes off the screen at the bottom, it will reappear at the top with the same horizontal displacement and still moving downward.

Taking the two's complement of a byte value

In the dazzle machine code, it is necessary to negate the value of DX and DY when the blob bounces off a wall:

```
DX = -DX     or     DY = -DY
```

Since the 6502 does not have a negate opcode, this must be implemented by taking the two's complement of the byte containing the value of DX or DY. This may be accomplished by using the following sequence of instructions:

```
LDA $FF          LDA $FF
EOR DX     or    EOR DY
STA DX           STA DY
INC DX           INC DY
```

4. EXPLORATIONS

• Experiment with allowing the user to control some of the parameters that are fixed in the implementation of dazzle in the text. One possible approach is to allow the paddles to be used to determine one or more of these values.

• What other actions might the blob take beside bouncing off walls and bending in midscreen that could lead to increased interest and variety of the produced displays?

• Modify Dazzle so that some of the parameters get modified in the inner loop.

LISTINGS

LISTING 20.1 DAZZLE MAIN PROGRAM

```
>LIST
    1 REM  ============================
    2 REM  =                          =
    3 REM  =      DAZZLE - VERSION 1  =
    4 REM  =        WRITTEN BY        =
    5 REM  = DR. RICHARD C. VILE, JR. =
    6 REM  =  ALL COMMERCIAL RIGHTS   =
    7 REM  =        RESERVED          =
    8 REM  =                          =
    9 REM  ============================
   10 WAIT=1000
   11 KBD=-16384:CLR=-16368
   15 DAZZLE=2048:FULL=-16302
   16 X=16:Y=17:DX=18:DY=19:BND=23
   17 BNDTIME=20:CHANGE=21:DBND=24
   72 LIST
  100 REM ================================
  101 REM =       M A I N   L O O P      =
  102 REM = REINITIALIZE FOR NEW DISPLAY. =
  103 REM ================================
  105 GR : POKE FULL,0
  106 COLOR=0
  107 FOR I=40 TO 47: HLIN 0,39 AT I: NEXT I
  110 RPT=0
  115 CHANGEVALUE=20+ RND (220)
  116 DBNDVALUE= RND (2)
  117 BNDTIMEVALUE=CHANGEVALUE/2
  118 POKE BNDTIME,BNDTIMEVALUE: POKE DBND,DBNDVALUE: POKE CHANGE,
      CHANGEVALUE
  119 POKE X,0: POKE Y, RND (24): POKE DX,1+ RND (3): POKE DY,1+ RND
      (3): POKE BND,0
  120 REM ========================
  121 REM =  I N N E R   L O O P  =
  122 REM =                      =
  123 REM = REPEATEDLY CALL THE ML =
  124 REM = DAZZLE CODE TO GENERATE=
  125 REM = THE GRAPHICS DISPLAY.  =
  126 REM = DURING EACH CALL THE   =
  127 REM = NUMBER OF BLOBS PLOTTED=
  128 REM = WILL BE EQUAL TO THE   =
  129 REM = VALUE OF THE VARIABLE  =
  130 REM = "CHANGEVALUE", WHICH   =
  131 REM = HAS BEEN POKED INTO THE=
  132 REM = ML DRIVER'S PAGE ZERO  =
  133 REM = VARIABLE "CHANGE".     =
  134 REM = BETWEEN CALLS CHECK FOR=
  135 REM = KEYSTROKES THAT CONTROL=
  136 REM = THE HALTING OF THE     =
  137 REM = DISPLAY, CONTINUATION  =
  138 REM = OF CURRENT PLOT, ETC.  =
  139 REM =                        =
```

```
 140 REM ========================
 145 IF (RPT MOD 2)#0 THEN COLOR=0
 146 IF (RPT MOD 2)=0 THEN COLOR=1+ RND (16)
 150 CALL DAZZLE
 152 DBNDVALUE= RND (2)
 153 POKE DBND,DBNDVALUE
 155 RC=0: GOSUB WAIT
 160 IF RC#0 THEN GOTO 105
 198 RPT=RPT+1
 199 GOTO 145
1000 REM ========================
1001 REM =    W    A    I    T    =
1002 REM =                         =
1003 REM = CHECK KEYBOARD FOR USER=
1004 REM = KEY.  KEYS STRUCK WILL =
1005 REM = INDICATE CERTAIN       =
1006 REM = DESIRES:               =
1007 REM =                        =
1008 REM =     0 - QUIT, RETURN   =
1009 REM =         TO BASIC.      =
1010 REM =  <CR> - WIPE DISPLAY   =
1011 REM =         AND START NEW  =
1012 REM =         ONE.           =
1013 REM =                        =
1014 REM = OTHERWISE: WAIT FOR    =
1015 REM =   NEXT KEY WHILE USER  =
1016 REM =   VIEWS DISPLAY.       =
1017 REM ========================
1020 KEY= PEEK (KBD)
1025 IF KEY<128 THEN RETURN
1030 POKE CLR,0
1035 IF KEY# ASC("Q") THEN 1040
1036 TEXT : CALL -936: END
1040 IF KEY#141 THEN 1050
1045 RC=1: RETURN
1050 KEY= PEEK (KBD): IF KEY<128 THEN 1050
1055 POKE CLR,0
1099 RETURN

>
```

LISTING 20.2 M.L. DAZZLE DRIVER

```
0000:              2 ;*********************************
0000:              3 ;*                               *
0000:              4 ;*     D A Z Z L E - M L         *
0000:              5 ;*                               *
0000:              6 ;*  MACHINE LANGUAGE DRIVER FOR  *
0000:              7 ;*  THE DAZZLE PROGRAM.  THIS    *
0000:              8 ;*  VERSION SUPPORTS THE MINIMAL *
0000:              9 ;*  SET OF DAZZLE FEATURES.      *
0000:             10 ;*                               *
0000:             11 ;*  WRITTEN BY                   *
0000:             12 ;*  DR. RICHARD C. VILE, JR.     *
0000:             13 ;*  JUNE 1979                    *
```

```
0000:             14 ;*  REVISED - JULY 1981          *
0000:             15 ;*                                *
0000:             16 ;**********************************
0000:             17 ;*
0000:             18 ;*
----- NEXT OBJECT FILE NAME IS LISTING 20.2 *****.OBJO
0800:             19          ORG   $800
F800:             20 PLOT     EQU   $F800
0010:             21 X        EQU   $10
0011:             22 Y        EQU   $11
0012:             23 DX       EQU   $12
0013:             24 DY       EQU   $13
0014:             25 BENDTIME EQU $14
0015:             26 CHANGE   EQU   $15
0016:             27 INDEX    EQU   $16
0017:             28 BEND     EQU   $17
0018:             29 DBEND    EQU   $18
0800:             30          ENTRY DAZZLE
0800:             31 ;*
0800:             32 ;*
0800:             33 ;**********************************
0800:             34 ;*      INITIALIZE CONTROLS       *
0800:             35 ;**********************************
0800:             36 ;*
0800:             37 ;*
0800:A9 00        38 DAZZLE: LDA   #$00
0802:85 16        39          STA   INDEX
0804:             40 ;*
0804:             41 ;*
0804:             42 ;**********************************
0804:             43 ;*    MAIN LOOP OF THE PROGRAM    *
0804:             44 ;* PLOTS POINTS FOR "CHANGE"      *
0804:             45 ;* REPETITIONS AND THEN RETURNS.  *
0804:             46 ;* DURING THIS LOOP, BOUNCING     *
0804:             47 ;* OFF THE EDGES OF THE SCREEN    *
0804:             48 ;* IS HANDLED AND BENDING IN THE  *
0804:             49 ;* MIDDLE OF THE SCREEN IS ALSO   *
0804:             50 ;* CONTROLLED.                    *
0804:             51 ;**********************************
0804:             52 ;*
0804:             53 ;*
0804:A5 11        54 LOOP:    LDA   Y
0806:A4 10        55          LDY   X
0808:20 00 F8     56          JSR   PLOT        ; PLOT X,Y
080B:A5 10        57          LDA   X
080D:18           58          CLC
080E:65 12        59          ADC   DX
0810:85 10        60          STA   X           ;X=X+DX
0812:A5 11        61          LDA   Y
0814:18           62          CLC
0815:65 13        63          ADC   DY
0817:85 11        64          STA   Y           ;Y=Y+DY
0819:             65 ;*
0819:             66 ;*
```

```
0819:             67 ;**********************************
0819:             68 ;* CHECK TO SEE IF BLOB IS STILL*
0819:             69 ;* INSIDE THE DISPLAY.  IF NOT- *
0819:             70 ;* ADJUST X,Y ACCORDINGLY AND   *
0819:             71 ;* CAUSE THE BLOB TO REFLECT OFF*
0819:             72 ;* THE BOUNDARY OF THE DISPLAY. *
0819:             73 ;**********************************
0819:             74 ;*
0819:             75 ;*
0819:A5 10        76          LDA  X
081B:10 03        77          BPL  NF1        ;IF X<0
081D:20 69 08     78          JSR  FIX1       ;THEN CALL FIX1
0820:C9 28        79 NF1:     CMP  #$28       ;IF X>=40
0822:30 03        80          BMI  NF2        ;THEN
0824:20 71 08     81          JSR  FIX2       ;CALL FIX2
0827:A5 11        82 NF2:     LDA  Y          ;IF Y<0
0829:10 03        83          BPL  NF3        ;THEN
082B:20 79 08     84          JSR  FIX3       ;CALL FIX3
082E:C9 30        85 NF3:     CMP  #$30       ;IF Y>=48
0830:30 03        86          BMI  NF4        ;THEN
0832:20 79 08     87          JSR  FIX3       ;CALL FIX3
0835:             88 ;*
0835:             89 ;*
0835:             90 ;**********************************
0835:             91 ;* BUMP BEND COUNTER.  IF BEND- *
0835:             92 ;* TIME IS EXCEEDED, THEN HAVE  *
0835:             93 ;* THE BLOB CHANGE DIRECTION IN *
0835:             94 ;* MID-SCREEN.                  *
0835:             95 ;**********************************
0835:             96 ;*
0835:             97 ;*
0835:E6 17        98 NF4:     INC  BEND       ;BEND = BEND + 1
0837:A5 17        99          LDA  BEND       ;IF BEND>=BENDTIME
0839:C5 14       100          CMP  BENDTIME   ;THEN
083B:90 11       101          BCC  TESTLOOP   ;
083D:A9 00       102          LDA  #$00
083F:85 17       103          STA  BEND       ;RESET BEND COUNTER
0841:A5 18       104          LDA  DBEND      ;IF DBEND<0
0843:10 06       105          BPL  YBENDS     ;THEN
0845:20 57 08    106          JSR  BENDX      ; CALL BENDX
0848:38          107          SEC             ;
0849:B0 03       108          BCS  TESTLOOP   ;
084B:20 60 08    109 YBENDS: JSR  BENDY      ;ELSE CALL BENDY
084E:E6 16       110 TESTLOOP: INC INDEX
0850:A5 16       111          LDA  INDEX
0852:C5 15       112          CMP  CHANGE
0854:90 AE       113          BCC  LOOP
0856:60          114          RTS
0857:            115 ;*
0857:            116 ;*
0857:            117 ;**********************************
0857:            118 ;* SUBROUTINES TO HANDLE BLOB   *
0857:            119 ;* REFLECTION OFF SCREEN EDGES  *
0857:            120 ;* AND BENDING IN THE MIDDLE OF *
```

```
0857:                  121 ;* THE SCREEN.                    *
0857:                  122 ;*********************************
0857:                  123 ;*
0857:                  124 ;*
0857:A9 FF             125 BENDX:   LDA  #$FF
0859:45 12             126          EOR  DX
085B:85 12             127          STA  DX
085D:E6 12             128          INC  DX
085F:60                129          RTS
0860:                  130 ;*
0860:                  131 ;*
0860:A9 FF             132 BENDY:   LDA  #$FF
0862:45 13             133          EOR  DY
0864:85 13             134          STA  DY
0866:E6 13             135          INC  DY
0868:60                136          RTS
0869:                  137 ;*
0869:                  138 ;*
0869:A9 00             139 FIX1:    LDA  #$00
086B:85 10             140          STA  X
086D:20 57 08          141          JSR  BENDX
0870:60                142          RTS
0871:                  143 ;*
0871:                  144 ;*
0871:A9 27             145 FIX2:    LDA  #$27
0873:85 10             146          STA  X
0875:20 57 08          147          JSR  BENDX
0878:60                148          RTS
0879:                  149 ;*
0879:                  150 ;*
0879:A9 00             151 FIX3:    LDA  #$00
087B:85 11             152          STA  Y
087D:20 60 08          153          JSR  BENDY
0880:60                154          RTS
0881:                  155 ;*
0881:                  156 ;*

*** SUCCESSFUL ASSEMBLY: NO ERRORS
```

SYMBOL TABLE SORTED BY ADDRESS

```
  14 BENDTIME          17 BEND          0857 BENDX          0860 BENDY
  15 CHANGE          N0800 DAZZLE         18 DBEND            12 DX
  13 DY               0869 FIX1         0871 FIX2           0879 FIX3
  16 INDEX            0804 LOOP         0820 NF1            0827 NF2
082E NF3              0835 NF4          F800 PLOT           084E TESTLOOP
  10 X                11 Y              084B YBENDS
```

SYMBOL TABLE SORTED BY SYMBOL

```
  10 X                11 Y              12 DX               13 DY
  14 BENDTIME         15 CHANGE         16 INDEX            17 BEND
  18 DBEND          N0800 DAZZLE        0804 LOOP         0820 NF1
0827 NF2              082E NF3          0835 NF4          084B YBENDS
084E TESTLOOP         0857 BENDX        0860 BENDY        0869 FIX1
0871 FIX2             0879 FIX3         F800 PLOT
```

INDEX

B

C

D

E

F

G

H

I

K

L

M

P

Q

R

S

T

V

W

apple II
PROGRAMMER'S HANDBOOK

This indispensable guidebook offers tips and techniques for effective programming in the four most common Apple programming languages.

Written specifically for the intermediate programmer, **Apple II Programmer's Handbook** is full of professional tips and techniques for programming in Integer BASIC, Applesoft BASIC, Apple(UCSD)Pascal, and 6502 Assembly Language. Each technique is illustrated with a complete program that will help you absorb higher-level structures and gradually work down to the more fundamental details. In fact, the author has simplified this "top-down" approach by incorporating programs that use subroutines and by presenting the same program in different languages.

With applications in graphics, education, utilities, languages, and entertainment, **Apple II Programmer's Handbook** is a must for anyone using an Apple computer. Whether you study the programs, run them as is, or modify them to suit your specific needs, this book explains everything you need to know to increase your understanding **and** enjoyment of programming your Apple computer, including information on:

- distinct features and capabilities of each programming language
- low-resolution graphics
- string arrays
- creating your own language
- the Apple Disk Operating System
- and more.

Richard C. Vile, Jr., is a senior software specialist for GemNet Software Corporation in Ann Arbor, Michigan. Active in the computer field since 1974, Mr. Vile has taught math and computer science at Eastern Michigan University, worked as a programmer, and written articles for several leading computer magazines.

GRANADA PUBLISHING
Printed in Great Britain
0 246 12027 4

net